Open Research Online

The Open University's repository of research publications
and other research outputs

A study of aspects of synchronisation and communication in certain parallel computer architectures

Thesis

How to cite:

Whitbread, Martin John (1989). A study of aspects of synchronisation and communication in certain parallel computer architectures. MPhil thesis The Open University.

For guidance on citations see FAQs.

© 1987 The Author

https://creativecommons.org/licenses/by-nc-nd/4.0/

Version: Version of Record

Link(s) to article on publisher's website:
http://dx.doi.org/doi:10.21954/ou.ro.00010129

Copyright and Moral Rights for the articles on this site are retained by the individual authors and/or other copyright owners. For more information on Open Research Online's data policy on reuse of materials please consult the policies page.

oro.open.ac.uk

UNRESTRICTED

A STUDY OF ASPECTS OF SYNCHRONISATION AND COMMUNICATION IN CERTAIN PARALLEL COMPUTER ARCHITECTURES

MPHIL SUBMISSION TO THE OPEN UNIVERSITY

1987

SUBMITTED BY MARTIN JOHN WHITBREAD

SUBJECT: COMPUTING

Date of submission: 28 September 1987

Date of award: 13 March 1989

ProQuest Number: 27775886

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent on the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

ProQuest 27775886

Published by ProQuest LLC (2020). Copyright of the Dissertation is held by the Author.

All Rights Reserved.
This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

ABSTRACT

This paper examines methods for synchronisation and communication between tasks in highly parallel arrays of processors. The development of various methods is researched and simulation techniques are applied to specific structures, to examine their effectiveness. Two approaches to simulation are presented, in the first case a discrete event simulator is applied to task synchronisation implemented with semaphores in a close coupled environment. Secondly the concurrent programming language Occam is used to simulate a systolic configuration of processors. In this case the design is verified, through actual system construction.

Conclusions are drawn regarding the design disciplines and structure imposed by the use of these simulation techniques. A close relationship is found between the behaviour of a simulation written in Occam and the same structure constructed from multiple processors.

Further research is suggested into the subject of dataflow processors, to find suitable means for simulating such systems, prior to implementation. A type of test vehicle is proposed that would operate a dataflow processor under the control of the development system.

CONTENTS

PART 2

SUMMARY

This thesis is divided into two parts which cover two approaches to the simulation of parallel computing systems. It also includes an introduction which reviews methods for categorising computer systems and specifically examines some that utilise concurrency as means towards gaining improved performance. Mechanisms for control and synchronisation in multiprocessor computer systems are examined.

In Part 1 of this thesis, some of the mechanisms described earlier in the introduction, are applied in a simulated tightly-coupled multiprocessor environment. Conclusions are drawn concerning the use of a discrete event simulator to model both a multiprocessor computer and the control/synchronisation algorithms used on it.

Part 2 of the thesis examines a method for simulating loosely-coupled architecture and uses the concurrent programming language Occam to simulate a systolic array. This array was constructed and its behaviour compared to the simulation in order to verify the use of a concurrent language as a design tool.

INTRODUCTION

1 AN EXAMINATION OF CONCURRENT COMPUTER SYSTEMS WITH SPECIFIC REFERENCE TO CONTROL AND SYNCHRONISATION METHODS

1.0 An Introduction to Parallel Processing

Parallel processing in computer systems is a means of exploiting concurrent events in the computing process. Concurrency implies parallelism, simultaneity and the use of pipelining techniques. Operations may take place in parallel, in different resources during the same time period, they may occur simultaneously, at the same instant, or they may be pipelined, occurring during overlapped time periods. Concurrency is a means of providing enhanced performance over and above that available from current technology. A number of parallel processing mechanisms have been developed in uniprocessor computers. They include :-

- A multiplicity of functional units
- Parallelism and pipelining within the CPU
- Overlapped CPU and I/O operations
- Use of a hierarchical memory system
- Balancing of subsystem bandwidths
- Multiprogramming and time sharing

The use of multiple functional units allows the CPU to carry out a number of operations in parallel. The CDC-6600 (1964)[1] has ten functional units [Fig 1.1] and utilises a 'scoreboard' to keep control of the utilisation of the CPU resources.

Pipelining techniques can be applied in the decoding and execution of instructions, to provide the overlapping of tasks by dividing them into subtasks. Each of these is executed concurrently by specialised hardware stages that function concurrently with other stages in the pipeline. [Fig 1.2]

The overlapping of CPU and I/O operations can be achieved using separate hardware to perform the I/O. These I/O controllers, channels or I/O processors can transfer data to and from memory using Direct Memory Access techniques. [Fig 1.3]

In order to close the performance gap between the fast CPU and relatively slow memory subsystems, a hierarchical memory system is often installed, with fast registers accessible by the CPU. Fast memory is used to provide a limited capacity cache, and so on through RAM and disk memory etc. to form a memory hierarchy that increases in capacity as it decreases in speed. [Fig 1.4]

As stated above, the bandwidth of a memory subsystem is likely to be less than that of the CPU. The same also applies to I/O subsystems. Various techniques are used to improve this imbalance, the IBM 3033[2] uniprocessor uses interleaved memory, where eight double words (each of 64 bits) can be fetched in one memory cycle from an eight-way interleaved memory system.

A more efficient use can be made of resources if the time occupied by a process during an I/O operation is utilised on the CPU by a different process. In a multiprogramming environment there are likely to be multiple processes competing for memory, I/O and CPU resources. Using time-sharing techniques, multiple processes can all be allowed equal access to the CPU resources, as oppose to multiprogramming where a priority process can occupy the CPU for an excessive period. [Fig 1.5]

The use of multiple processors, in a wide variety of configurations, is now an increasingly common solution to a mismatch between system performance and system requirements. In some cases a limited number of general purpose computing systems are linked so that they can co-operate in the execution of an algorithm. At the other extreme a large number of special purpose processors may be applied to a specific or a narrow range of tasks, such as image processing. The wide range of past, current and proposed computer architectures has led to a variety of classification schemes which attempt to allocate all types to one of their categories.

1.1 Architecture Classification Schemes

Flynn[3] has proposed a scheme that classifies computing systems into one of four types depending upon the number of instruction and data streams present:-

* Single instruction stream, single data steam (SISD)
* Single instruction stream, multiple data stream (SIMD)
* Multiple instruction stream, single data stream (MISD)
* Multiple instruction stream, multiple data stream (MIMD)

SISD computers are conventional single processor systems in which advantage may have been taken of a number of architectural features such as pipelining to improve performance. SIMD machines are array processors, where each instruction operates on an individual data set from a distinct data stream, rather than a single operand. All processors of this type receive the same instruction but operate on different data sets from different data streams. Vector processors are usually included in the SIMD

classification. MISD computers are not generally considered very useful structures. Some architectures are difficult to place in Flynn's scheme and as new types are developed these difficulties increase.

Other classification schemes have been proposed. Feng's scheme[4] (1972) is based on serial versus parallel processing. That by Shore[5] (1973) is an extension of this approach and Shore defines six different categories of machines [Table 1]. Several major architectural types are excluded from Shore's taxonomy, which mainly serves to subdivide Flynn's SIMD class (and includes SISD configurations as Machine Type 1).

Table 1 Shore's Taxonomy

Machine 1 :

This category is represented by conventional von Neumann architecture with a single control unit (CU), processing unit (PU), instruction memory (IM) and data memory (DM). A single data word is read in parallel from the DM. The PU may contain multiple functional units which may also be pipelined. This category includes pipelined scalar (e.g. CDC 7600)[6] and pipelined vector computers (e.g. CRAY-1)[7].

Machine II:

The DM fetches a bit slice from all the words in memory instead of the Machine I approach of reading all the bits of one word. The PU operates in bit-serial fashion. (e.g. ICL DAP)[8].

Machine III:

Combining features of Machines I and II, this category of machines can read either words or bit slices. (Sanders Associates OMEN-60 series) [9].

Machine IV:

In this machine the PU and DM of Machine I are replicated as multiple PEs. There is no direct communication between the PEs, only through the control unit. (e.g. The PEPE Machine) [10].

Machine V:

As an extension of Machine IV, PEs are arranged in a 1D array, with nearest neighbour connections. A PE can address data in its own or nearest neighbour's memory. (e.g. ILLIAC IV) [11].

Machine VI:

Representing a different conceptual approach, where the logic is an integral part of memory, a so-called logic in memory array (LIMA). This has been applied experimentally to image processors (e.g. the PixelPlanes machine) [12].

Handler[13] has proposed a classification scheme for identifying the degree of parallelism and pipelining built into the hardware structures of a computer system. Parallel-pipeline processing is considered at subsystem levels:

- Processor control unit
- Arithmetic logic unit
- Bit-level circuit

A computer can be defined in Handler's classification using a triple containing six independent entries. These describe the number of processors [K] and the number that can be pipelined

[K'], the number of ALUs [D] and the number that can be pipelined [D'], and the word length in a PE [W] and the number of pipeline stages in all PEs[W']. Thus the TI ASC[14], with one controller linked to four arithmetic pipelines, each with 64-bit word lengths and 8 stages, gives:-

TI-ASC = <1x1,4x1,64x8> = <1,4,64x8>

from a general case of:-

T(c) = <KxK',DxD',WxW'>

Handler's scheme does not allow for variable numbers of stages in different functional units (W'), or variable pipeline chaining (D'), as can be found in the Cray-1.

1.2 SIMD Architectures

A synchronous array of parallel processors is known as an array processor, consisting of multiple processing elements (PEs), operating under the control of a single control unit (CU). Thus an array processor can be classified as an SIMD machine because it can handle single instruction and multiple data streams. Vector processors are sometimes placed into this category. Array processors make use of multiple PEs operating in lock step, in parallel.

There are two basic architectural organisations for SIMD computers, array processors using random access memory and associative processors using content addressable memory. An early array processor was Slotnick's "Solomon" which became the Illiac IV, with multiple PE's connected in a square array. Others that have been developed include the Burroughs Scientific Processor (BSP)[15] and Parallel Element Processing Ensemble (PEPE)[16], and the Goodyear Aerospace Staran[17] and Massively

Parallel Processor (MPP)[18]. Of these, the BSP and the MPP are direct developments from the Illiac IV, PEPE and STARAN are associative array processors. The topology of the PE array in an SIMD processor can vary in complexity from a linear array to hypercube structures. (Feng [19]).

Vector processors contain a number of high speed arithmetic pipelines capable of processing data vectors in response to a single instruction. When operating on long vectors these systems exhibit high performance but with short vectors and anything more than a minimal scalar content in the algorithm, the performance falls off rapidly (Bashkow[20]). They are now often regarded as being outside of Flynn's classification.

1.3 MIMD System Types

MIMD machines consist of several processors executing different code and processing different data but their operation forms part of a common algorithm. These systems vary in the homogeneity of the processors present and in the degree to which they are coupled. In a tightly-coupled system the number of processors is fixed and they all operate under a strict control scheme. This control is usually centred in a hardware unit. Tightly-coupled MIMD computers consist of a number of processors, memory modules and I/O channels, connected through a set of three interconnection networks. Every processor may be connected to every memory module and I/O channel through cross-bar switches. [Fig 1.6 (a)] Alternatively the processors, memory and I/O may be interconnected by a common bus structure.

Loosely-coupled MIMD computers [Fig 1.6 (b)] communicate through some form of message transfer system to pass information between processors. This arrangement is most efficient when the interaction between tasks is minimal. The following sections examine MIMD architecture in more detail as a precursor to the simulation of a tightly-coupled MIMD system and the simulation and construction of a loosely-coupled MIMD system.

Fig. 1.1 The System Architecture of the CDC-6600 Computer

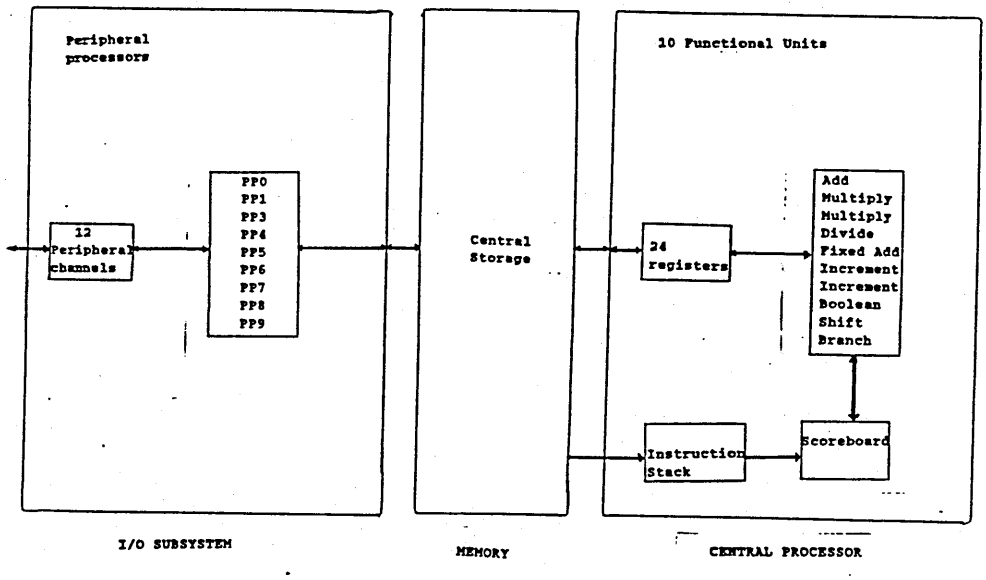


Fig. 1.2 A Pipelined Processor

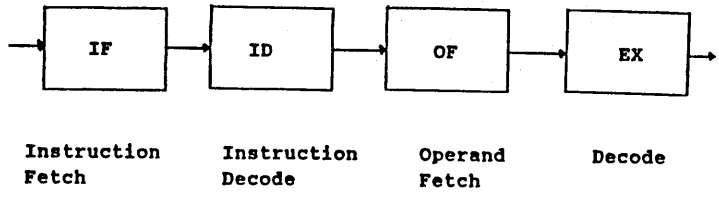


Fig. 1.3 I/O Using Direct Memory Access

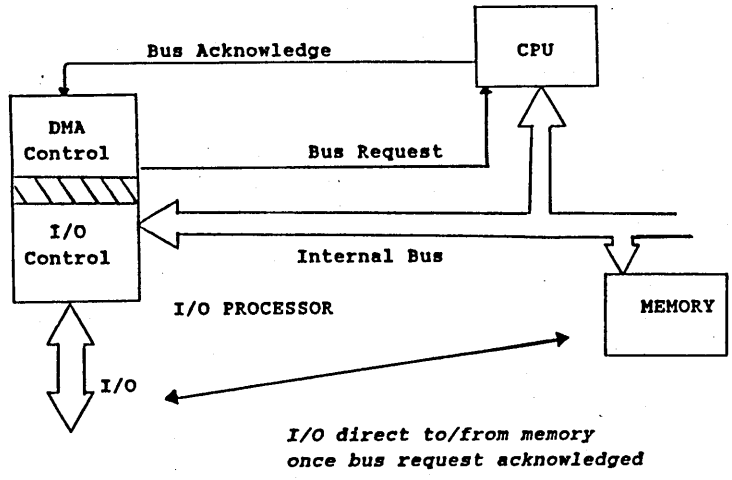


Fig. 1.4 Memory Hierarchy

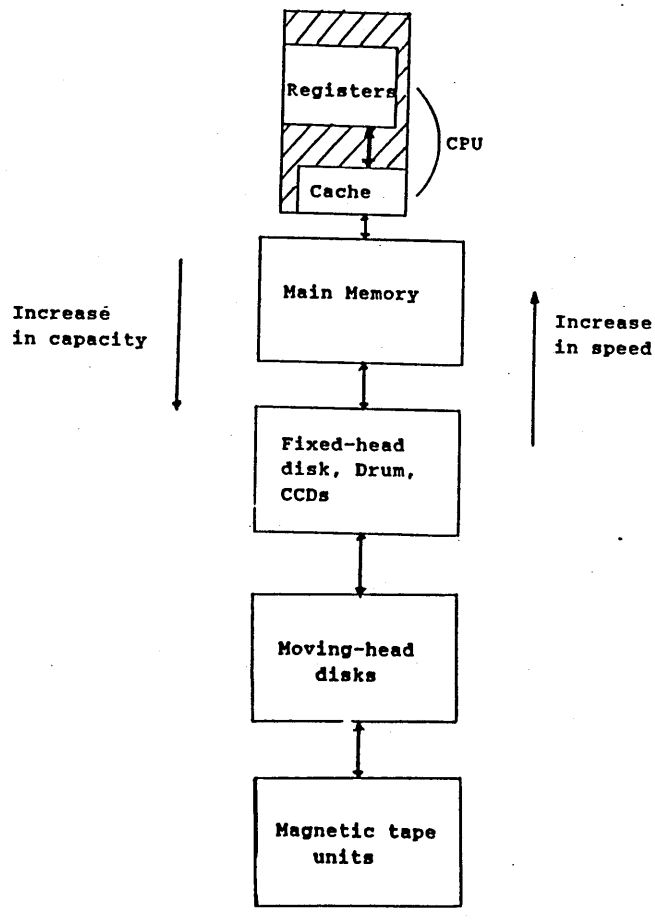


Fig. 1.5 Operating System Approaches to Acheive Parallel Processing in a Uniprocessor Computer [Hwang & Briggs]

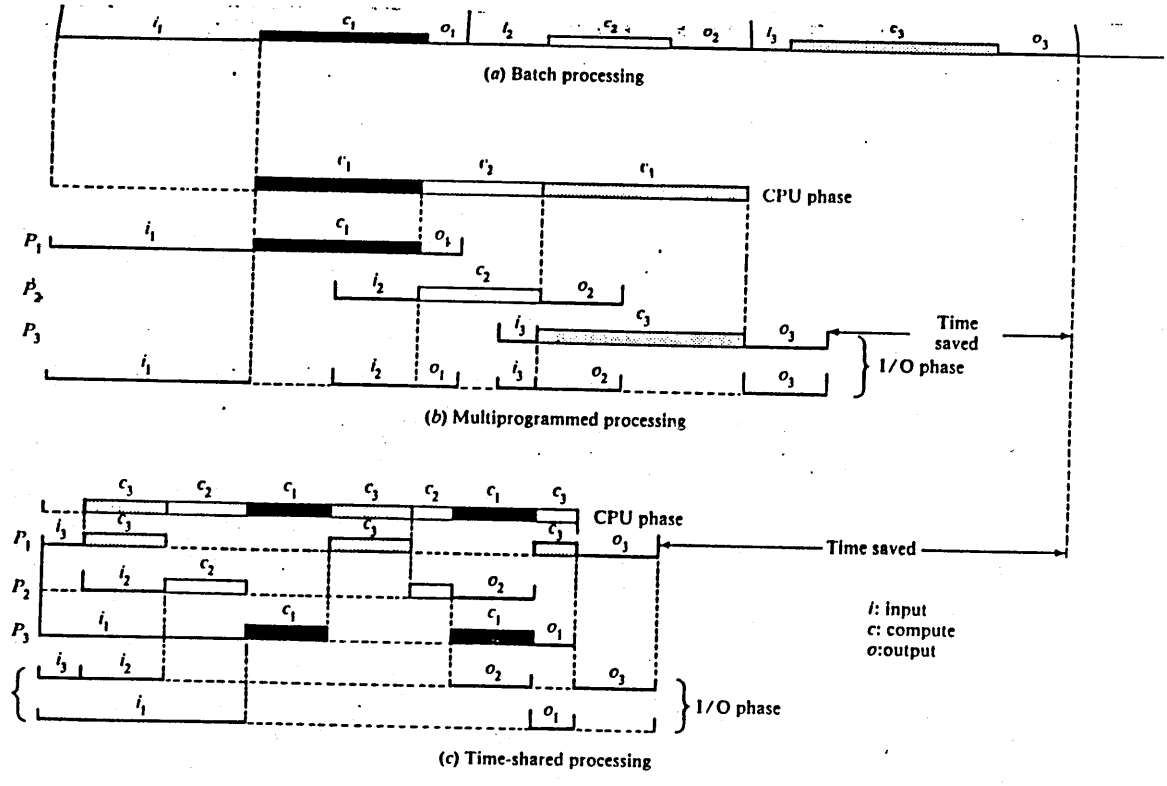


Fig. 1.6 a)

A Tightly Coupled Configuration

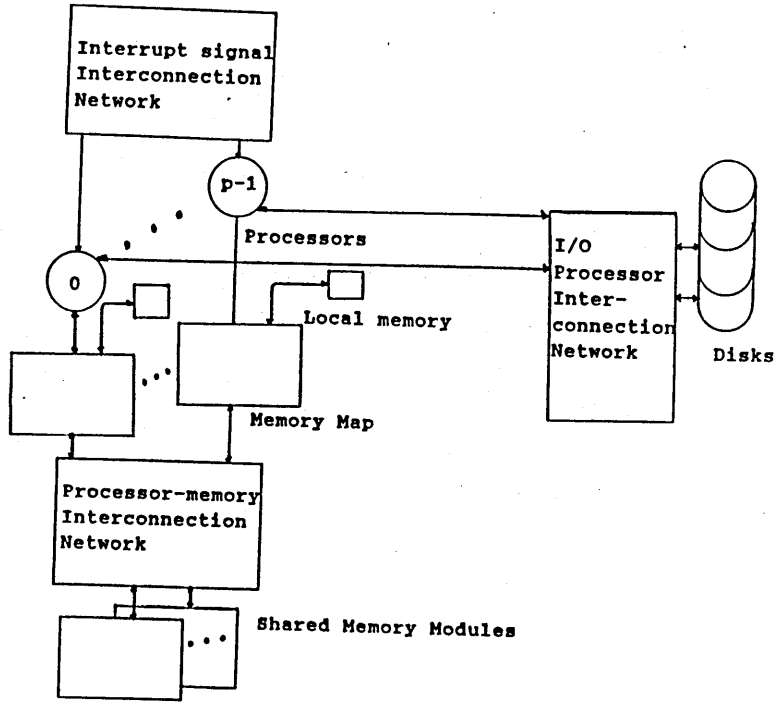
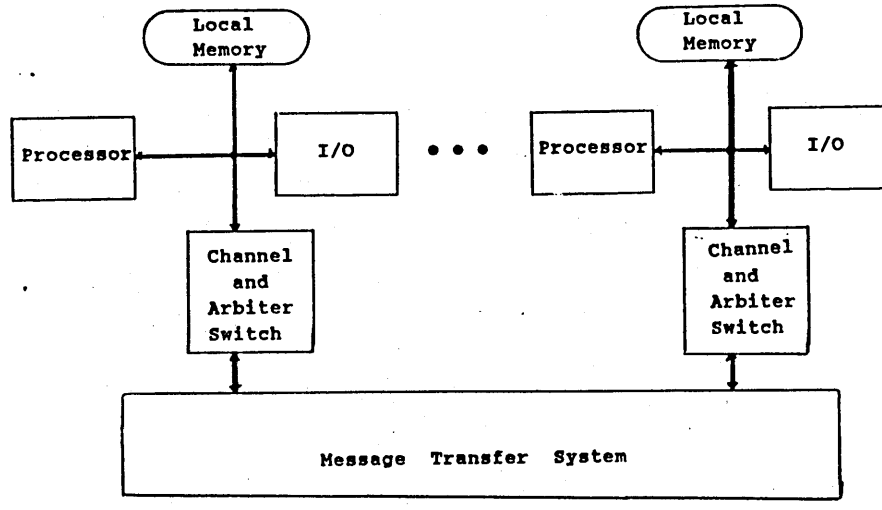


Fig. 1.6 b) Loosely Coupled Computer Modules



2 MIMD ARCHITECTURES

The remainder of this Introduction is concerned with characteristics of MIMD systems. It is therefore necessary to examine in more detail the types of architecture used in implementing MIMD systems in order to appreciate the problems encountered and their possible solutions. Relevant aspects of MIMD architecture are discussed below.

2.1 Integrity in MIMD Systems

Many multiprocessor systems are implemented using processors not originally designed for multiprocessing. The integrity of a processor can never be assumed to be absolute. A processor forming a node in a multiprocessor system may fail at any time. That failure may well be the cause of deadlock in the system, if other processes are relying on the results. A facility to retrieve the system state from a failed node, possibly using an externally accessible register file, can be desirable in order to avoid having to restart the algorithm on a reconfigured processor array. The system would require some form of deadlock detection in order to be able to respond to processes in a 'deadly embrace'. This is particularly so where the deadlock is caused not by a failed node (which could be detected by other means) but by the incorrect ordering of interprocess communications, creating a situation where two or more processes are all waiting for each other.

2.2 Tightly-coupled Systems

Tightly-coupled multiprocessor systems can be configured around cross-bar switches or a single bus. The latter is one of the simplest ways that a tightly-coupled MIMD system can be arranged. Having processors accessing common memory does not necessarily mean degradation of access time to local memory, which may be on the same bus card or accessed across a local bus. The VME bus specification [21] includes a local memory bus, VMX, which allows local access by processors to memory and memory mapped I/O, so avoiding contention on the VME bus itself.

Reliability and throughput can be increased by providing multiple buses. The question of bus control immediately arises, since multiple processors will be needed to order their access to the bus.

2.2.1 Priority Schemes in Bus Connected Architectures

A static priority scheme, based on a daisy-chain arrangement of access permission, has been used to allow one of a number of devices access to the bus. The highest priority device can lock out all other devices which are lower down the chain, in order of priority.

Designs have been implemented using a high speed (CSMA/CD) network interface as a 'serial backplane'. Dynamic priorities can then be implemented following a message collision. This is done by decreasing the delay period before a retry is made to access the network, if the node continues to fail to gain access. Once access is made the retry period drops back to the original longer period. Similar dynamic priorities can be established on bus systems using bus controllers. Algorithms for dynamically permuting priorities such as a least-recently-used scheme, where

the priority of a device is increased until it gains access, can provide a fairer share of the bus to all potential users than a fixed priority scheme. Simple bus controllers can provide this type of facility.

A more complex algorithm is the first-come-first served scheme which provides for very efficient use of the bus but requires the storage, in order of arrival, of requests made for access to the bus. Requests made must be timed and allowance made for the simultaneous arrival of two requests and their subsequent ordering. Other schemes include polling, where a single device makes a request either on a single open-collector bus request line or a dedicated signal path to a bus controller.

Once the bus is free, request is acknowledged and a bus busy signal exerted. The device polls the bus controller, which deals with the request when it can.

The relative efficiencies of the different schemes vary and are likely to be a compromise of cost against complexity. One of the least efficient schemes is that of allowing a fixed time slot on the bus for each processor, resulting in empty slots where the processor has no immediate demand for the bus.

2.2.2 Communication Using Switched and Multiport Memory

In an MIMD system, where a separate path is available for every memory unit so that there are sufficient paths to satisfy the demands made for them, performance is limited by the bandwidth-speed product of the individual access paths alone. In such a system a complex crossbar switch must not only be able to maintain connectivity but also be able to deal with simultaneous requests for the same resource, on a priority basis. The

complexity and cost of a crossbar switch may limit the number of processors and memory modules that can be interconnected.

By applying switching and priority arbitration to the memory interfaces a multiport memory is created. The use of multiport memories can limit the size of the system as there is little scope for increasing the number of ports, hence access paths, to a multiport memory once implemented. There are also multiple pathways to the memory modules to consider, making it impossible to use a commercial backplane.

A system based on a crossbar switch centralises the cost and complexity of the hardware in that switch, which may also need to use redundancy to protect against failure. As the system is expanded, overall performance can be expected to increase. A modular implementation of the switch permits degradation but not collapse of the system in the event of failure. The cost of a crossbar switch grows as the square of the number of paths. There is a trade-off in the various choices for tightly-coupled interconnection between multiple processors and memory modules. The lower cost solutions will introduce conflicts and result in delays to process execution. Extremes of complexity may not be worthwhile if the application does not warrant the related development time and cost.

2.2.3 Deadlocks During Resource Allocation

Dijkstra [22] has termed a state where two processes are halted, waiting on each other for an event, as a 'deadly embrace'. Multiple processes operating in a computer system are invariably sharing resources. Resources include processors, memories, I/O and bulk storage. Resource allocation must be performed in such a way that the minimum time is wasted

satisfying the associated request. The situation may be dynamic, particularly in terms of demand for memory. A single process will run on a processor, generating processes for other processors. These may require access to any memory module in the system. In order that a deadly embrace can occur, one of the following conditions must hold :-

1) A process needs exclusive control of some resources.

2) A process needs to keep control of resources while further resources are sought.

3) Processes hold resources that are required by others.

2.2.4 Deadlock Avoidance in MIMD Systems

A deadlock (or deadly embrace) can be avoided if :-

a) A suspended process does not retain control over a resource. It can be made to release it, to retrieve it later.

b) A single once-and-for-all request is made for resources, no dynamic demands being allowed.

c) A cyclic ordering of resources takes place to avoid a circular chain of ownership.

Deadlocks that do occur often result from a quite reasonable local requirement. These local requests will inevitably clash unless they are placed under a form of global supervision. Reyling [23] suggests the use of a resource allocation processor to accept requests from the processors in the system and allocate resources. Keedy [24] suggests the extension of PV semaphores to include a word in which bits are allocated to represent a set of resources. The requests need to be lodged with the resource controller and the requesting processes must wait for their request to be satisfied. The control of resources using a

dedicated processor makes the survival of the system dependent on that processor. An improved scheme takes the form of floating control which moves between processors in the system. At any time one (and only one) processor has control of the resources.

The uniqueness of control can be maintained using a single line which is pulled high by the controlling processor. Other processors interrogate this line to determine whether a processor has control.

Where the systems each have a separate supervisor program, resource sharing must take place through shared files which again have to be protected from access conflicts. The replication of supervisor code on every processor is a large overhead on memory requirements. Should one of these processors fail it would be difficult to perform a restart as the status of the system needs to be remapped so that the process can be continued on another processor.

2.3 Loosely-Coupled Systems

There is not the same degree of memory conflict experienced by loosely-coupled systems as in tightly-coupled configurations. Each processor has local memory and I/O. Processes executed in loosely-coupled systems communicate by exchanging messages. The bandwidth of the communication system and the volume of traffic passing over it, will affect the performance of the system as a whole. Some form of switching is likely to be needed, to route messages from computer module to computer module. For the system to be efficient communication between tasks needs to be minimised. The interface between the computer modules and the message transfer system often contains a channel and arbiter

switch. These will buffer messages in a high speed communication memory accessible to all processors.

If two or more simultaneous requests occur, the arbiter selects one for service. The message transfer system can be a simple timeshared bus or a multiported memory system. The message throughput is critical to the system performance.

3 CONCURRENT PROCESS CONTROL STRUCTURES

Concurrent processing presents problems of control and synchronisation between processes. These are discussed below and later [Section 4] some are examined in a simulated multiprocessor system.

3.1 The Synchronisation of Concurrent Processes

In a multiprocessor environment, where separate but related processes are running, there is likely to be a need for process synchronisation. Both synchronisation and communication facilities may be provided at operating system or instruction set level. Specific facilities are needed to allow the efficient operation of the hardware on which concurrent processes are running, to prevent problems of deadlock, lockout or incorrect synchronisation between processes occurring.

There is a significant difference between tightly-coupled and loose-coupled systems in their respective requirements for process synchronisation. For tightly-coupled systems the semaphore mechanism has been developed. In loose-coupled systems the unbuffered "channel" acts as both as a synchronisation mechanism and as a rendezvous point.

3.1.1 Semaphores

The development of semaphore schemes for concurrent synchronisation has been accompanied by attempts to solve a number of closely related problems associated with the mutual exclusion of processes from data areas and critical code sections. The simplest form of process synchronisation involves just two processes and a single integer variable accessible by both processes, a semaphore. In the proposal by Dijkstra a

semaphore is a variable that can have a zero or signed integer value equalling the number of process activations required. One process signals another to activate using the semaphore as a means of communication between them. A negative condition is possible, indicating the number of waiting processes.

If semaphores are constructed as program or operating system functions there is bound to be some loss of efficiency associated with their use. A more efficient implementation is in hardware or microcode. In order to cause the controlling semaphore variable to decrement or increment, two commands are needed. These are defined as P (Wait - Dutch nomenclature) and V (Signal), causing the semaphore to decrement or increment respectively.

A semaphore that is restricted to zero or unity is known as a binary semaphore. If larger integer values are allowed then the term counting semaphore is used. This type of semaphore was used in the simulation of a multiprocessor environment, reported later . A common use for P and V is to permit concurrent processes to exchange data during execution. The localisation of the semaphore in commonly accessible memory implies that the system is tightly-coupled.

Concurrent processes wishing to communicate in a tightly-coupled environment can do so through memory buffers. The contents of a buffer needs to be controlled so that the consumer does not attempt to read data from it until some data is present. A control variable is used in the manner of a semaphore; when the variable equals zero, no data is present and no access to the buffer should be made.

Consumer processes wishing to access the buffer will be required to wait if the producer process has not yet output any data. If the buffer is one message deep then the producer and consumer will need to rendezvous in order to exchange longer items of data. This is found in Ada synchronisation primitives[25] and Occam channels[26].

A producer-consumer solution can be implemented in Concurrent Pascal [Ben-Ari][27]:-

In the program shown below information is generated by the procedure 'producer' and appended to a buffer. This data cannot be used by the procedure 'consumer' until the integer n changes to a positive value. The execution of signal(n) by the producer will cause n to be incremented. The consumer will wait only if n has a zero value. Both producer and consumer are executed concurrently in the main program. The counting semaphore n is initialised to zero.

```
program producer-consumer;
var n: semaphore;
procedure producer;
begin
     repeat
          produce;
          append;
          signal(n);
     forever
end;
procedure consumer;
begin
     repeat
          wait(n);
          take;
          consume;
     forever
end;
begin (* main program *)
     n := 0;
     cobegin
          producer; consumer
     coend
end.
```

The 'append' and 'take' sections of code are adding to and taking from the message buffer. If they are mutually exclusive, further semaphores will be needed for access control.

A circular buffer (e.g. a UNIX pipe) sits between the producer and consumer processes. Using two semaphores, one for the number of occupied buffer words and another for the number of free words, PV primitives can be applied to communication between the producer and consumer processes.

Two instructions, corresponding to P and V are provided in the ICL 2900 instruction set[28]. These respectively increment a main memory location, setting a condition code on the result and conversely, set a condition code on the state of the memory location before decrementing it. The MC 68000 microprocessor instruction set contains a binary semaphore instruction, BSET[29], a combined bit test and set instruction.

3.1.2 Synchronisation in Loosely-Coupled Systems

Where there is not a commonly accessible memory for processors to communicate through i.e. in loosely-coupled systems, other problems arise. When process A is running on a separate processor from process B but wishes to communicate with it, process B may not be ready to listen. Conversely B may be expecting data that has not yet been sent. This problem seldom arises in close-coupled systems where buffers can be constructed in common memory locations. Only where the system runs out of buffer space will the same problem occur.

In loosely-coupled systems, communicating through I/O or dedicated channels, there may be a need for synchronisation between processes in order to establish communication.

Communication controllers with access to DMA channels can receive or transmit messages free of CPU intervention. Messages would then be placed in local memory buffers. Where communication is direct to the CPU and concurrency extends to the local software environment, it is possible for the processor to continue to do useful work while a concurrent routine completes a communication process with another processor. With increased levels of integration, it is possible to provide communications processors on the same chip as the CPU. The Inmos Transputer[30] has four serial channels which can be activated by writing a single word to the selected channel. Because the channel mechanism is unbuffered, the sending process must wait for the data to be received. This has the effect of synchronising both sending and receiving processes.

If I/O buffering is needed it has to be written into the software, and buffering processes run concurrently. This feature of the Occam language, released as the basic language for the Transputer, is discussed and utilised later in Part Two.

3.2 Implementation of Semaphores

The introduction of the semaphore by Dijkstra provided a tool from which more powerful primitives could be constructed. A semaphore, expressed as an integer variable is altered only by calls to procedures wait(S) and signal(S).

Their function is summarised as :-

Wait(S) : If S > 0 then S := S - 1, otherwise the execution of the process that called wait(S) is suspended.

Signal(S) : If some process P has been suspended by a previous wait(S) on the semaphore S, then wake up P, otherwise S := S + 1

The operation of these primitives on the same semaphore must be exclusive. If they occur simultaneously, they are executed one at a time, but in an unpredictable order. Just which process is woken by 'signal' is not defined, assuming there is more than one process suspended on the semaphore(S).

3.3 Mutual Exclusion Between Concurrent Processes

If two processes P1 and P2 need to be mutually exclusive in a critical section of code in each processor, binary semaphores can be used to provide a solution. An example of this mechanism in Concurrent Pascal (Ben Ari) is :-

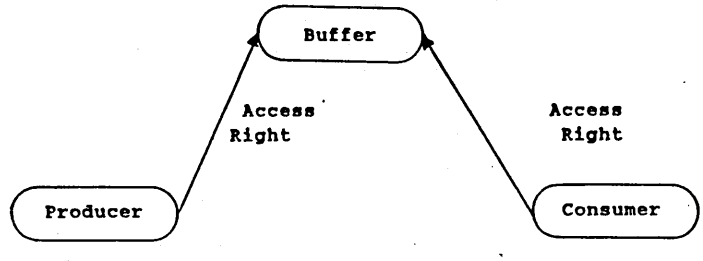
```
program   mutualexclusion;
var s : semaphore;
procedure p1;
begin
     repeat
          wait(s);
          crit1;     (*Critical Section*)
          signal(s);
          rem1;      (*remainder of process*)
     forever
end
procedure p2;
begin
     repeat
          wait(s);
          crit2;
          signal(s);
          rem2;
     forever;
end
begin (* main program *)
     cobegin
          p1; p2     (* concurrent execution of p1 and p2*)
     coend
end.
```

The two critical regions crit1 and crit2 have been made mutually exclusive, execution being controlled by wait and signal primitives. Only one process is allowed in the critical section at one time. When more than one process is allowed into a critical section the semaphore s can be allowed to take higher positive values. In this case the semaphore is initialised to the maximum number of processes allowed. The semaphore is decremented to 0 as those processes enter the section and is returned to the original value when they all leave it.

3.4 Interprocess Communication Using Monitors

In his paper first describing Concurrent Pascal, P. Brinch Hansen[31] introduced the concept of monitors. These define a shared data structure which can be used by processes for communication. A monitor can synchronise concurrent processes and transmit data between them. Two processes communicating do so through buffers to which they both have access. The access rights of these processes are defined in an access graph. [Fig 1.8]

Fig 1.8



This is an access graph showing that the processes can use the buffer. A process consists of a private data structure and a sequential program. One process cannot operate on the private data of another process.

Monitors provide a shared data structure and the functions to access the data. They contain the critical section of code needed for access to that data. While a semaphore is an elegant low level primitive, its use in complex systems is dangerous, incorrect use producing errors that only occur following a given pattern of events. The monitor is designed to allow concurrency while retaining the advantages of a structured construct.

3.5 Process Rendezvous in Loosely-Coupled Systems

The semaphore and monitor constructs are closely related to systems with common memory space, (which have been defined as close-coupled). Where the memory areas are separate these constructs are difficult to implement. Hoare introduced the concept of a rendezvous between processes in his paper "Communicating Sequential Processes" (CSP)[33]. The implementation of process to process communication in CSP as described by Kuo, Linck and Saadat [34] shows a clear relationship between CSP and the concurrent language Occam which is utilised later to model a concurrent system.

In both CSP and Occam a process can output a variable in the manner :-

```
Producer                          Consumer
x:= 3                             prod:= 1
A! x ..........................>  A? val
P:= x + 1                         i:= 1
```

Neither process can proceed until the unbuffered transfer of the single message has occurred. In CSP communication is from process to process, in Occam it is through individual channels, defined in each process.

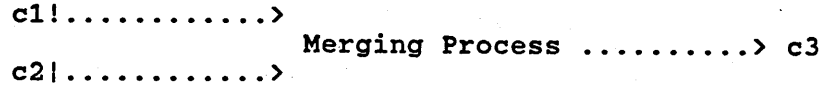
The rendezvous causes both processes to coincide, whichever arrives at the rendezvous point first must wait for the other. A rendezvous is also available in Ada using the 'accept' statement which waits for a call from another process, which may occur before or after the statement is executed. The two processes rendezvous in order to synchronise or exchange data.

3.6 Guarded Commands

In several implementations of concurrent languages, including Ada and Occam, alternative sections of code can be selected according to the result of a "guard" expression. An example in Occam is :-

```
WHILE TRUE
 VAR X:
 ALT
  c1? x                --Guard (input from Channel 1)
     c3! x
  c2? x                --Guard (input from Channel 2)
     c3! x
```

This program reads either channel c1 or c2, whichever is ready with input and acts as a merging process for the two inputs. Data is then placed on channel c3. If both c1 and c2 are ready for input, only one of the guarded processes is selected :-

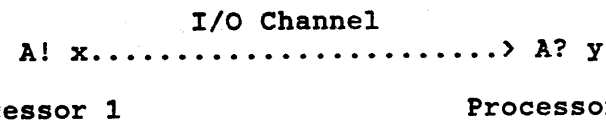


3.7 Language Selection

The semaphore and monitor constructs apply to concurrent programming environments where the multiple processors are coupled through common memory. Languages such as Concurrent Pascal are oriented towards this type of environment. The rendezvous found in the Ada accept statement and Occam channel matches these languages to a loosely-coupled architecture where processes are communicating through a message transfer system.

The Occam channel is a straight forward structure to implement in hardware or software, depending on the level of concurrency present in the system.

The output of A!x can be directed across on a data channel to another processor. Thus:-



The rendezvous is a natural by-product of the need for handshaking in the transfer of data between processors. The implementation described in Part 2 of the thesis performs this handshaking in software, both processors must therefore rendezvous before communication takes place.

The Occam language was applied to the modelling of a multiple processor array, which was subsequently built and the characteristics of the hardware and software model were compared.

4 SYSTOLIC ARRAY ARCHITECTURE

The throughput of conventional Von Neumann processors is often limited by the speed of the I/O. Even specialised signal processors which often have very fast multipliers included in their design, are limited in application by restricted I/O throughput. For tasks involving relatively simple computation but a great deal of I/O, such as image processing, the single processor is unlikely to provide enough throughput.

4.1 Computation vs. I/O Time

Kung[38-41] has postulated a number of systolic designs based around an inner product processing element. The potential for systolic arrays, in terms of raw Mips is illustrated by Fig. 4.1 & 4.2 :-

Fig. 4.1

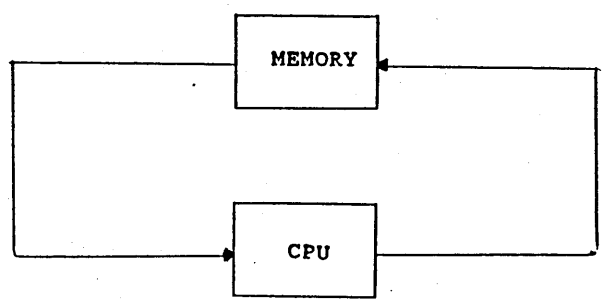
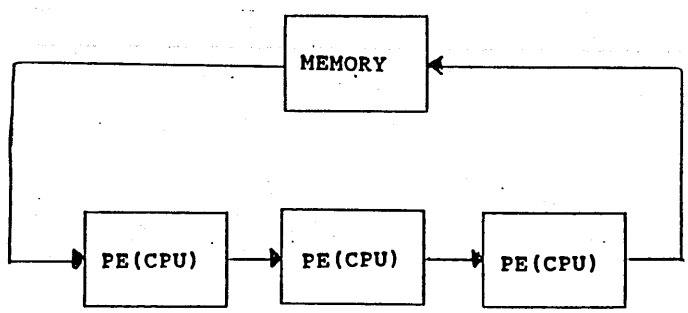


Fig. 4.2



Where multiple processors are used the system is not limited by memory bandwidth because data is only extracted and replaced once per item. However multiple computations have been carried out on each item as they pass through the pipeline of processors.

Orders of magnitude improvements of this kind are only possible if multiple computations are performed per I/O access. The nature of the problem and the resulting demands on internal storage are important factors. Computation and memory structure have to be arranged to give a suitable balance between computation time and I/O time.

4.2 Arrays of Processors

Processing Elements (PEs) can be configured in a variety of ways. Three options for mesh connections (neighbour-to-neighbour communication) are linear, orthogonal and hexagonal. Fig. 4.3

The internal communication paths are regular and easily implemented. At the boundaries external connections to memory or other special purpose processors are established. Some boundary processors only input a zero value, while others have no output paths. An alternative to a mesh-connected arrangement is to provide some degree of broadcast communication. This is classified as a semi-systolic array and can take one of several forms according to the nature of the broadcast data.

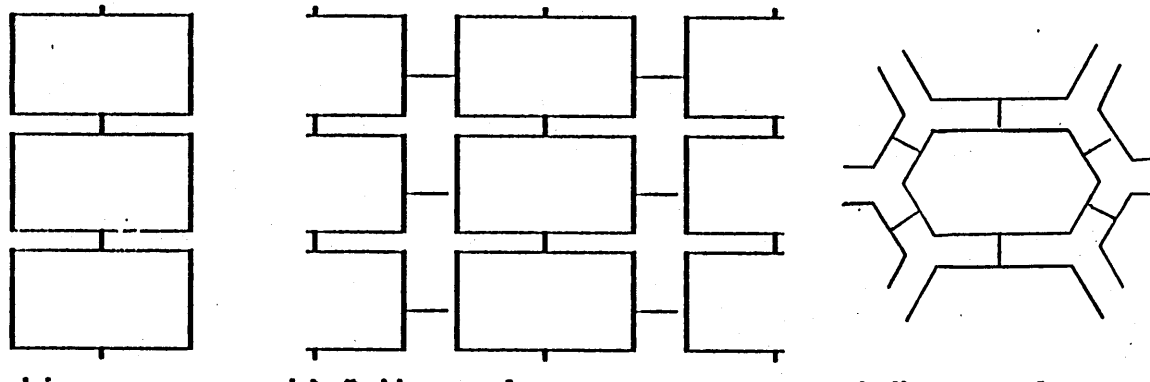
Systolic arrays can be effectively utilised in algorithms where two matrices are to be multiplied. An example of this is image convolution (See Section 7.4) where a small filter array is used to change a larger array representing image pixels. The filter array contains values (weights) which affect the image in different ways when various patterns and weight values are used. The algorithm requires each pixel input Xi to be multiplied by each of K weights. The results for each pixel are combined and a new image array is produced.

If each item of data from the image array is to be multiplied by each of k weights, then either the inputs or the

results may be broadcast and either the weights or results moved systolically through the array. Fig. 4.4 a)-c)

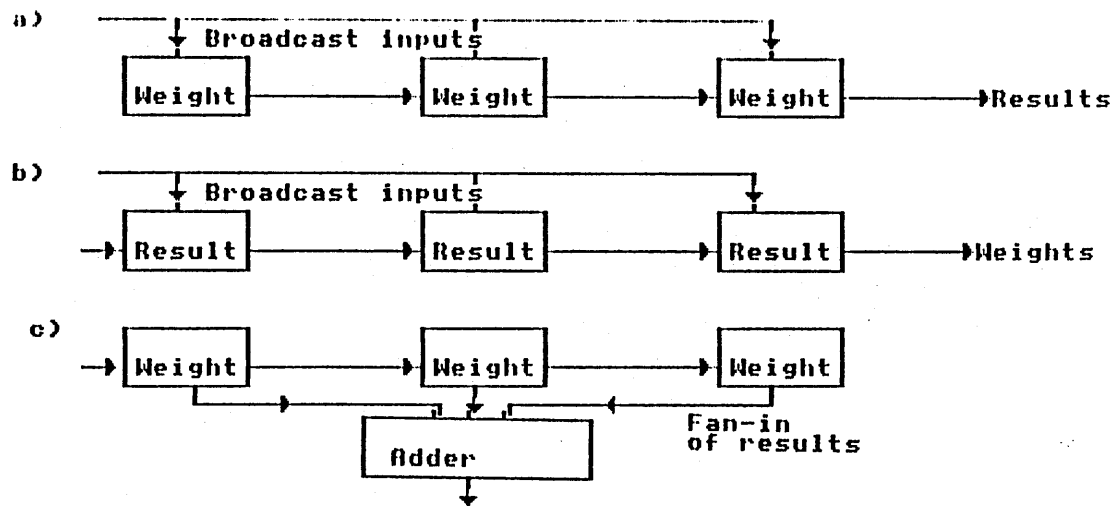
Pure systolic convolution is achieved if no global communication is used. For reasons of design convenience and transmission speed this approach is often preferred. Designs for pure systolic convolvers can again hold the results or the weights in the Processing Element with the information moving in a different direction, or in the same direction at a different speed. The objective in both cases is to move the data and weights across each other, in order that every item in the data array (image) is multiplied by every item in the weight array (filter). Fig. 4.5 a)-d)

Fig 4.3



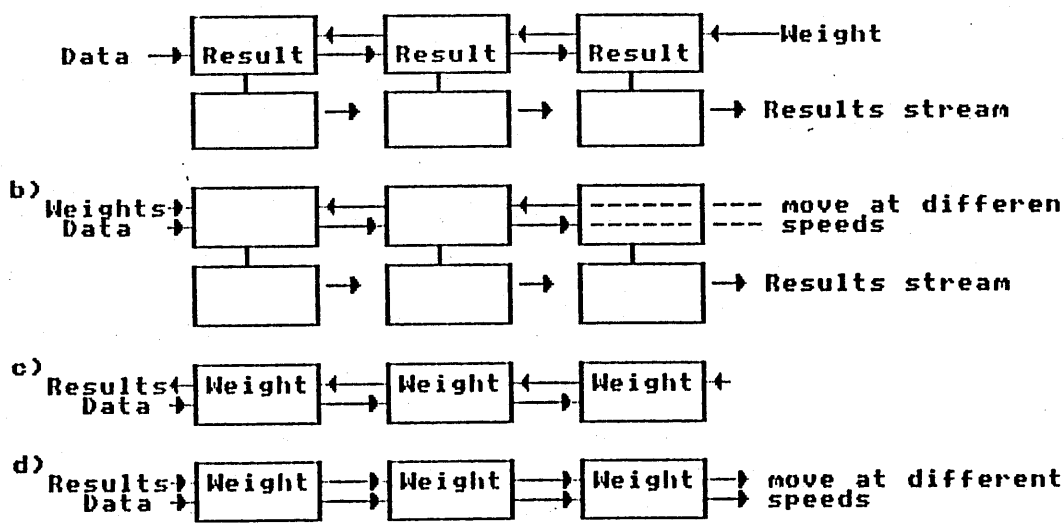
Each Processing Element (PE) is connected to two or more neighbours and data passes systolically between them.

Fig 4.4 a)



The entry in each box indicates the data item that remains stationary.

Fig 4.5 a)



Where results remain stationary a separate results stream is needed to extract data.

Other combinations are of course possible. Where weights remain static provision may have to be made to transmit the weights systolically before the data stream appears. A degree of complexity within the PE would allow for much greater flexibility in the kind of application to which the systolic array could be applied e.g. convolution and correlation. Internal registers containing weights could be loaded systolically (or broadcast) from a controlling system or the PE might be a fast processor, locally controlled. This would involve hardware, software or microcoded arithmetic functions in the processor. Various architectures have already been used in the construction of systolic arrays. S.Y.Kung[42] has defined asynchronous systolic arrays as wavefront arrays. The problem of global clocking is avoided by data-driven synchronisation and Kung introduces a conversion of signal flow graphs into data-flow graphs. This is achieved by replacing delay operators controlled by the global clock with a separator locally controlled by handshaking.

4.3 1D Arrays

A 1D array of PEs has some appeal because of the simplicity of the design. A single dimensional row of processors, with a single data entry point and a single results exit point, connected as a pure systolic array, can be applied to a wide range of problems.

A 1D convolution system with a fixed weight, can be constructed:-

Given a vector of signals X = (Xi), i = 1,2,...n

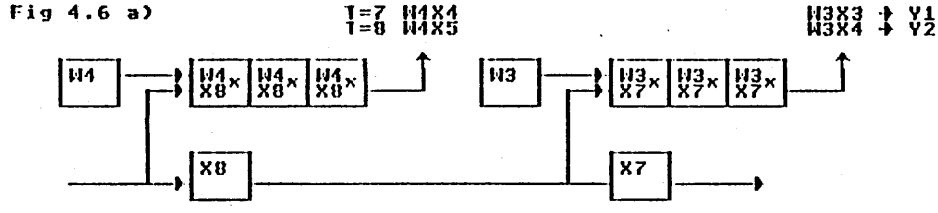
and another vector of weights W = (Wj), j = 1,2,...k

with K << n, the signal X is convolved with the kernel W to compute:-

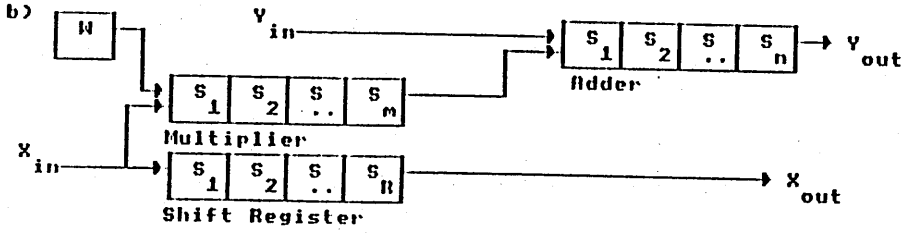
$$Y_s = \sum_{i=0}^{k-1} W_{i+1} \cdot x_{i+s}$$

H.T.Kung describes a method which, if the kernel remains static instead of sliding over the signal, utilises pipelining to cause a skew in the data and results streams so that the information(results) arriving at an adder from a previous PE is in alignment with the locally derived result. Fig. 4.5 a) This gives a general case; Fig. 4.6 b)

Fig 4.6 a)



b)



Where M = No of pipeline stages in the multiplier

A = " " " " " " adder

R = A + 1

(The same skew can be accomplished by using a 2-stage shift register for data transfer)

4.4 Multidimensional Convolution

The convolution of a 2D array(image)

$$X = \{Xij\},\ i = 1,2,.....,m$$

with a 2D kernel $W = \{Wij\}$, $i = 1 \rightarrow k$, $j = 1 \rightarrow p$, where $k << m$ and $p << n$, is to compute:-

$$Yrs = \sum_{i=0}^{k-1} \sum_{j=0}^{p-1} Wi+1,j+1 \ . \ Xi+r,j+s$$

$$\text{for } r = 1 \rightarrow (m-k+1) \text{ and } s = 1 \rightarrow (n-p+1)$$

The first result (Y11), is generated by placing the kernel over the image such that W11 covers X11, multiplying the corresponding elements of W and X, and summing these products. It is then necessary to slide the kernel one position to the right in order to generate Y12. Each element in each row is computed by scanning a row at a time, until Wkp covers Xmn.

The noise filtering of images is one application and, because it produces demonstrable results, is used here as an example of the convolution algorithm.

4.5 ND Convolution using 1D Systolic Arrays

If a 2D kernel is applied to a 2D data array, the requirements for a 1D solution can be illustrated as shown:-

	1	2	3	4	5	6	7
1	[1/1	1/2	1/3]	1/4	1/5	1/6	1/7
2	[2/1	2/2	2/3]	2/4	2/5	2/6	2/7
3	[3/1	3/2	3/3]	3/4	3/5	3/6	3/7
4	4/1	4/2	4/3	4/4	4/5	4/6	4/7

Where the 3*3 boxed elements in the array are those currently under the kernel. Scanning the array serially we have:-

1/1, 1/2, 1/3, 0, 0, 0, 0, 2/1, 2/2, 2/3, 0, 0, 0, 0, 2/1, 2/2, 2/3, 0, 0, 0, 0, 3/1, 3/2, 3/3.

Moving the kernel to the right will produce correct results as long as it does not overflow the edge of the data array. For a large array, e.g. a 2D image, the processing of these boundaries on the basis of a 'wrap-round' of the kernel is probably acceptable as the object of interest in the image array is unlikely to be so near to the edge of the array.

It is obvious that an m*n image convolution with a k*p kernel requires kp PEs, if every computation is to be performed concurrently in a separate PE. After every p cells there is an additional delay of (n-p) to account for that part of the data not under the kernel at that time. This is added to the delay of A+1 present in every cell. If there are large numbers of zeros in the kernel, the related cell can be eliminated by causing an additional cycle of delay in the previous cell. This eliminates an unnecessary overhead of A cycles of delay for every zero kernel element. The use of the pipelined delays has two purposes, firstly to align the array under the kernel and secondly to ensure that each pixel is fetched from the memory into the array only once. The use of linear memory to store the original (2D) array is consistent with this arrangement. A DMA fetch of the data can be carried sequentially without recourse to a complex mapping algorithm.

4.6 Multiplication in Bit Level Systolic Arrays

The type of algorithm used in systolic arrays applied signal processing commonly consists of multiply-accumulate-normalise computations. Early work in applying bit-level systolic arrays to multiplication was carried out at RSRE Malvern by McCanny and McWhirter[43]. The simple case of two 2-bit binary numbers A and B is illustrated below. Multiplication occurs in such a manner that the partial products are located to contribute appropriately to the product bits.

If :-

A = 11

B = 11

```
      B
    1   1
  -------
    1   1  |  1
           |     A
    1   1  |  1
```

A 4 bit partial product is produced:-

1 0 0 1

Each product bit is obtained by :

a) Taking the first bit of the product of the relevant bits from A and B (Pi = 1 if Ai and Bi = 1)

b) Summing Pi with any previous contribution to that order bit, including carries from the next lower order bit.

c) Generating a carry Ci to input to Pi+1.

It is possible to define functions needed to perform this algorithm a:

sum_out = (sum_in) EXC.OR (A.B) EXC.OR (carry_in)

and

carry_out = (A.B)AND(sum_in)OR(A.B)AND(carry_in)
OR(sum_in)AND(carry_in)

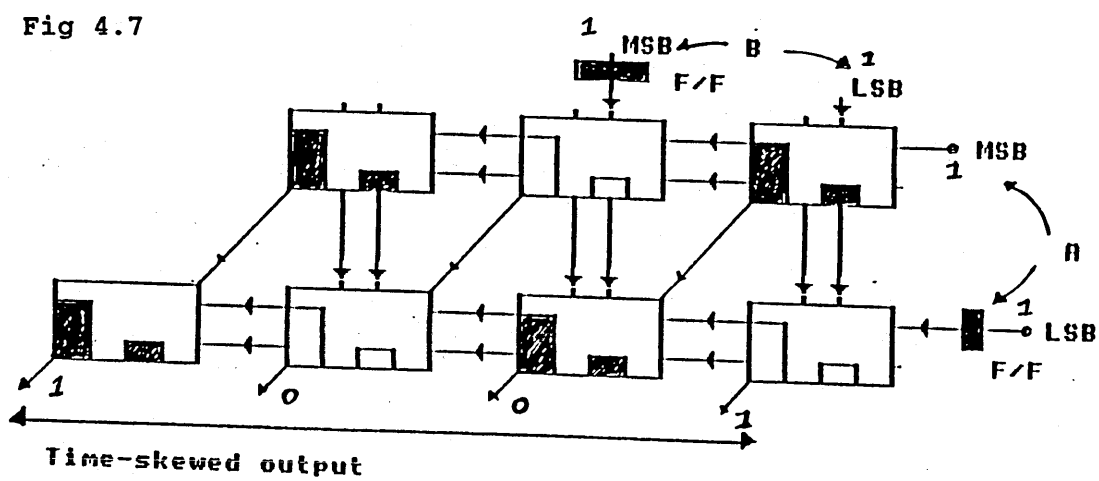
Where all the outputs are latched on a common clock. The Boolean relationships described use only a few gates. McCanny and McWhirter have suggested that these processor elements can be combined together to realise a systolic multiplier. Additional processors are needed to handle carries, and flip-flops to correctly sequence the input:- Fig. 4.7.

For n-bit words, the basic array needs n*n processors, plus ((n*n)/2)+(n/2) processors for the carries. This gives a total of:-

(n*n) + (n*n)/2 + n/2 = n/2.(3n+1)

Additional circuitry is needed to time-skew the incoming data.

Fig 4.7



4.7 Matrix x Vector Multiplication

Using word-organised, linear systolic arrays a matrix times vector multiplication can be performed:-

Fig. 4.8 a)

$$\begin{matrix} Y_1 \\ Y_2 \\ Y_3 \end{matrix} \quad * \quad \begin{matrix} W_{11} & W_{12} & W_{13} \\ W_{21} & W_{22} & W_{23} \\ W_{31} & W_{32} & W_{33} \end{matrix} \quad = \quad \begin{matrix} X_1 \\ X_2 \\ X_3 \end{matrix}$$

If the words are sliced into bits then a bit level systolic array can be used to process the operation :

Fig. 4.8 b)

$$\begin{matrix} W_{11} & W_{12} & W_{13} \\ W_{21} & W_{22} & W_{23} \\ W_{31} & W_{32} & W_{33} \end{matrix} = 2^{(n-1)} \begin{matrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{matrix} + 2^{(n-2)} \begin{matrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \\ \vdots & & \\ 1 & 1 & 0 \end{matrix} + 2^{0} \begin{matrix} 1 & 1 & 1 \\ 1 & 0 & 1 \end{matrix}$$

(bit 0)

4.8 Correlation

The vector of Fig 4.8 a) can assume the character of a reference pattern of bits for a correlator or pattern matcher. A typical application is for extracting known digitally-coded reference patterns from received signals. The reference pattern needs to be time-skewed for the output Y to be computed :-

$$Y = A_{ref.} * x$$

Corry and Patel[44] have described a 64-stage bit-slice correlator, arranged as a 1 bit reference, 4 bit data device.

4.9 Sorting

The process of sorting is widely used in image processing to find the median value of a localised group of pixels. Hooper[45] describes a systolic array for sorting, composed of two types of cell, a delay cell and a sort cell. The latter consists of a gated magnitude comparator and a crosspoint switch. Two serial input streams enter a sort cell, initially the crosspoint switch is set in the default direction. Comparisons of the two bit streams will result in either the switch remaining at default or switching. The sort array implements a bubble sort algorithm, the larger numbers being routed in one direction, smaller numbers in another. Successive application of the sort cells will cause at each stage, larger numbers to 'bubble-up' and smaller numbers to 'sink down'. The sorting takes place serially and is pipelined at bit-level. The throughput of the median filter is determined by the delay through a sort cell. Once the pipeline is full the pixel delay becomes this cell delay times the pixel word length.

4.10 Implementations

The design of systolic arrays in a programmable format is a compromise between cost and performance. For arrays implemented on a single VLSI chip there are problems of the amount of memory available to each cell. The GEC GRID[46] processor array provides a 32 word x 1 bit Two-port RAM for the storage of intermediate results. The NCR GAPP[47] array provides 128 bits of RAM for each cell but neither of these arrays has provision for accessing additional memory and are limited in application to tasks such as image convolution. Both the CMU WARP[48] processor and the Inmos Transputer[49] can access external memory and thus extend their application range. The latter is limited in I/O bandwidth by serial data transfers and it seems likely that the design is intended for later integration. The CMU Warp cell is a powerful processor using 16 bit data busses and is based around the Weitek 32-bit floating point multiplier and alu chips. Control is provided by an AMD 2910A microsequencer and the cell has only limited local address generation facility, data memory addresses are broadcast systolically.

THE THESIS - PART 1

5 THE EXAMINATION OF SOME CONCURRENT CONTROL MECHANISMS BY COMPUTER SIMULATION

This section describes the progressive construction of a simulation program representing a tightly-coupled multiprocessor system. This is used later as a test bed on which to examine and verify some of the control structures that are applied in close-coupled multiprocessor systems.

The computer simulation of various types of concurrent control mechanisms provides many benefits, not least that of easy modification. An algorithm can be run with different input streams, to evaluate its performance. Functions may be introduced which could be implemented by hardware, software or microcode. These may eventually become part of the instruction set or need to be designed into the operating system. Any implementation is improved if the algorithm that is developed is both powerful and yet simple in its structure. Excessive complexity can lead to inefficient or even an incorrect implementation of the algorithm, particularly if it is installed by the user.

The following section traces the development of a simulated system on which various multiprocessing control mechanisms were to be applied.

5.1 Timing Aspects

A simulation of the control algorithm in a multiprocessor system must be able to maintain precise control over its

representation of time. It should be possible to introduce a stream of events, arriving at discrete intervals of time. These events, representing tasks to be scheduled and controlled by the algorithm under examination, are best traced throughout their existence if they have separate identities, run times and other attributes.

5.2 Task Queues

The queues used in control algorithms must be represented and used in such a way that their contents are properly controlled. Tasks must leave queues and enter others when the algorithm demands. Queues and resources need to be shown as finite entities and there needs to be control over attempts to enter full queues or buffers, as well as over the possession of resources.

5.3 Limitations Imposed by the Simulator

Any simulation is necessarily subject to the limitations of the methods used. It may not be possible to simulate certain common occurrences in the environment of the system under examination without a disproportionate amount of effort and distortion of the simulated system. The representation of time by the simulator is important for its own efficiency. If we assume that no events occur between system clock periods then the choice of simulator type is narrowed. A discrete event simulator provides facilities for the simulation of concurrent events. These events are controlled by a system clock and no events are assumed to occur between clock pulses.

The presence of a single global clock to control events

introduces the requirement that the actual level of simulated events must be decided and anything occurring below that level can be assumed to proceed without need for further simulation. Events relating to bus contention, occurring at a lower level than a process scheduler under examination, would be ignored. Only successful bus transfers would be of interest.

5.4 The Method Used

Using a discrete event simulator, an environment was devised that would allow a scheduling algorithm based on Brinch Hansen's 'Shortest Job Next' algorithm[35], to be evaluated. The intention was to develop a robust model to which features could be added later in order to investigate various control mechanisms. The synchronisation of tasks, the application of semaphore techniques and the effects of different input streams were all to be applied to the model. These are described in detail below. This initial experimentation was carried out, not to evaluate Brinch Hansen's algorithm, but to match the problem to the simulation language being applied and to identify any shortcomings. The language ECSL (Extended Control and Simulation Language) [36] was applied to the problem, running on a Honeywell 66 mainframe computer in batch mode.

5.5 The Model

The model was intended to represent a system composed of both hardware and software. A number of empty slots in memory were assumed to be available for the execution of tasks in a multiprocessing environment. The model assumed that the physical transference of tasks or processes to appropriate memory buffers took place immediately that places became available.

A multiport memory was assumed, to which all execution nodes (processors) had access, forming the buffer into which all tasks selected for activation must pass. At any time a processor can retrieve an allocated task from the buffer for execution. It does so without contending for access with any other processor that might be retrieving a different task. The generalisation that all processors have access to all processes was not applied. Scheduling activity was viewed as centralised, allocating tasks that were generated (or activated) by another, undefined activity. The scheduler was assumed to operate with a queue of waiting processes which was added to by external activities.

The use of a simulation language was found to impose both limitations and structure on the organisation of the model. Compile time checks helped to produce a robust structure. The language ECSL was chosen after a prolonged search for a suitable simulation medium. Multi-tasking systems were considered, using memory bank switching, but intertask communications were limited to messages being passed through files, which restricted their usefulness. All program descriptions shown later match the actual ECSL code used and this code is given alongside the description.

5.6 The Simulated Environment

In order to verify a scheduling algorithm by simulation it was necessary to make a number of assumptions about the environment in which it would operate. If a variety of different loading's were to be placed on the algorithm, to verify its correctness, then the simulation should provide sufficient flexibility over time parameters. The scheduler needed to be placed between a source of generated tasks (input) and a sink for tasks ready for consumption by available processors:-

Task
Generation.....> Scheduler .....> Consumption

(Source) (Sink)

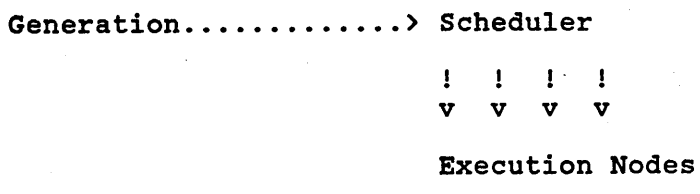
The arrival times of generated tasks needed to be controlled so that the arrival intervals were within a specified range. Then, by extending or shortening this range, the input load could be measured. This generation function was designed to produce an identifiable, and hence traceable, output. This allowed the order of execution to be observed and the reasons for any unjustified overtaking of high priority tasks by lower priority tasks made traceable. The incoming tasks needed an identity, attributes of priority, duration and an empty slot to contain the identity of the execution node (when that stage was reached). These were all easily implemented in ECSL.

Execution nodes were duplicated in the model and were treated as members of sets. They were occupied by specific tasks for the duration of the task's runtime. An identifiable task was attached to each node. As the model was developed,

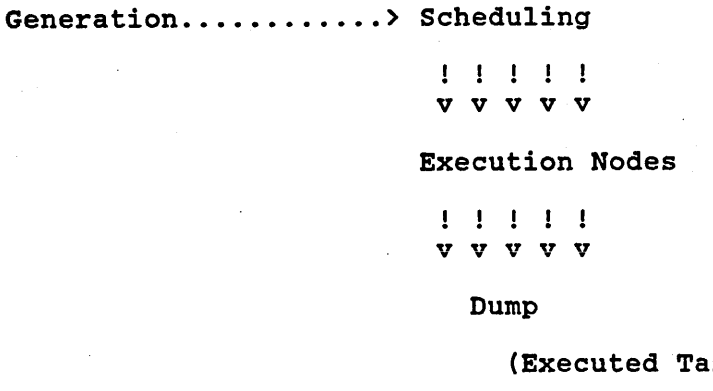
synchronisation semaphores were introduced, to allow the suspension of waiting tasks, which were then moved into an 'inactive' queue.

5.7 The Model Structure

The model now takes the form :-



Once the tasks had been completed on an execution node, they needed to be removed from that node and placed, for diagnostic purposes, into a 'dump', (an executed task queue). This was to ensure that no tasks were lost during the simulation run. Now the structure of the model is :-



5.8 Task Generation

The generation of tasks consisted of a process involving their assignment, complete with priority and runtime values, to a queue of ready-to-run tasks. This queue was searched, on a priority basis and not length of run, as in the original algorithm. The actual length of run used in Brinch Hansen's algorithm was considered to be dependent on factors not available prior to execution. Run-length was allocated as a fixed value, randomly selected, at task generation.

Selection by priority allowed dynamic changes to be made to the order of selection. Tasks moved from their point of origin into the queue of waiting process :-

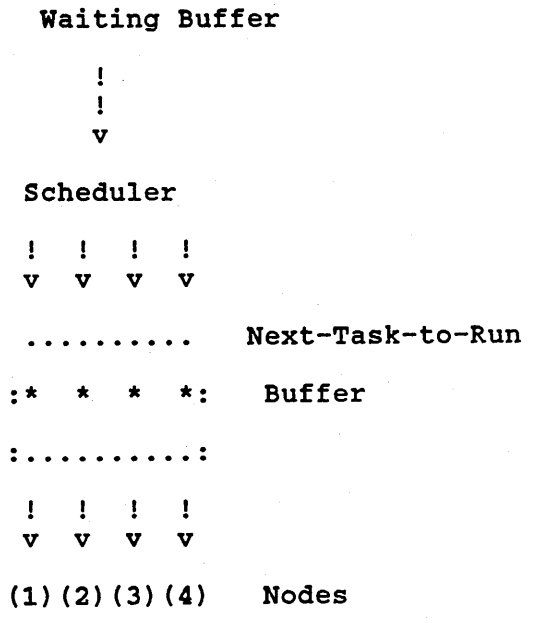
```
Origin.................> Task No. '5'
                         Priority '2'
                         Runtime '10'
                         Node     '?'     =  Queue of waiting
                         ----------          tasks
                         Task No.
                         Priority
                         etc.
```

This was the principal activity of the task generator. Whenever it became time for this routine to be run (at controlled intervals) a task was pulled from the set Origin, allocated the values required by the scheduler and put into the queue of waiting processes. As entry to this queue depended upon the frequency of running the generation routine, long intervals limited the numbers waiting in the queue, restricting or even

eliminating priority criteria. This sometimes resulted in low priority tasks passing through an empty queue, while subsequent higher priority tasks got stuck in the queue.

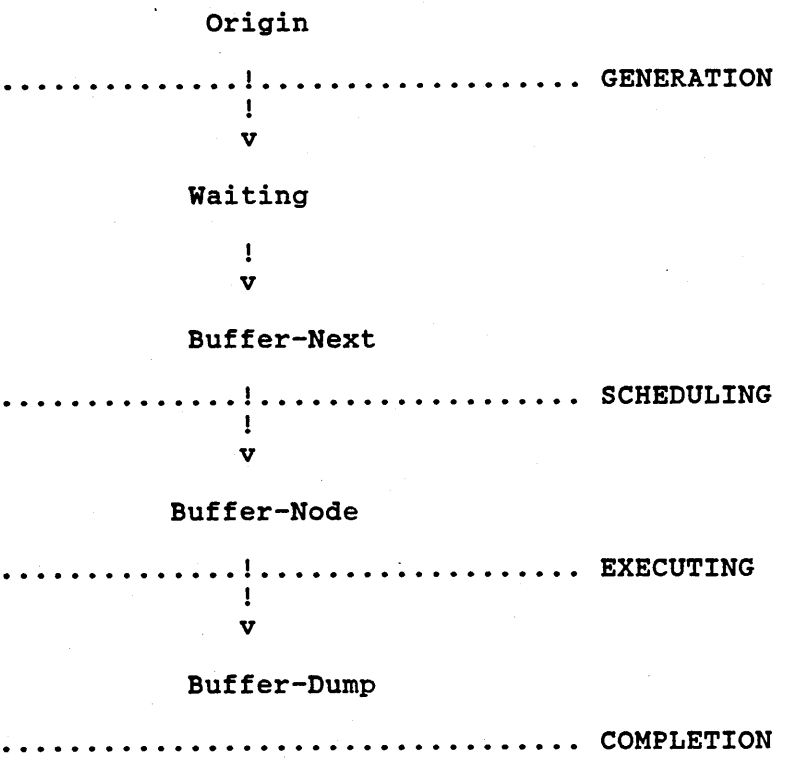
When tasks were queued in the buffer, selection by priority could take place. With a low arrival rate the buffer remained empty and tasks were executed on a first come first served basis irrespective of priority.

Each node was supported by a Next-Task-To-Run buffer, eliminating any delay in loading a new task:-



Selection into the Next-Task-to-Run buffer (Next) from the Waiting queue was on a priority basis and was carried out if there was a task to be chosen and a vacant buffer slot. ECSL provided a condition statement for testing empty queues, which greatly facilitated this feature.

Tasks passed through four distinct phases, between their origin and their completion:-



5.9 The Scheduling Algorithm

There were two operations inherent in the scheduling algorithm, to retrieve the task with the lowest priority number (highest priority task) from the Waiting queue and to put it into a 'Next' buffer, should there be one empty. (It was assumed that a cross-bar switch existed between the nodes and Next buffers). Nodes were allocated on a one-for-one basis and the condition :-

No of slots in Next > No of Nodes

would have provided no tangible benefit. The second operation allocated the task to a vacant node.

The description of the scheduling algorithm is given below, alongside the ECSL listing. This part of the simulation involved the transfer of a task from Waiting into Next :-

PROGRAM STEPS	LISTING
Start	BEGIN RELEASE
Test time (Always true	TIME OF RELEASE LE 0
interval after 1 clock period)	CHAIN
Test vacancy	B LE 5
B <= 5	
Find lowest	FIND TASK I IN
priority task	WAITING
Transfer task	TASK I FROM
to NEXT	WAITING
Increment vacancy	B = B + 1
counter, B= no used places in NEXT	
Set Interval Counter = 1	TIME OF RELEASE = 1
Do forever	REPEAT

There was now at least one task in NEXT which could be allocated to a node if one was free:-

PROGRAM STEPS	LISTING
Test time interval	BEGIN ALLOCATE TIME OF ALLOCATE LE 0 CHAIN
Task in NEXT and a Node Free?	EXISTS NODE IN FREE AND EXISTS TASK IN NEXT
Select task (first) from NEXT	FIND FIRST TASK H IN NEXT
Select node (first) in free	FIND FIRST NODE N IN FREE
Allocate node to task	EXEC OF TASK H = N
Activate node	NODE N FROM FREE INTO ACTIVE
Move task into USER	TASK H FROM NEXT INTO USER
Allocate RunTime	TIME OF TASK H = RUNTIME (H)
Decrement count of used buffer spaces	B = B -1
Reset time interval	TIME OF ALLOCATE = 1
Do forever	REPEAT

5.10 The Task Expiry Algorithm

Tasks that had completed their RunTime period in USER needed to be located and removed from the node on which they ran. For diagnostic purposes they then passed into an additional buffer DUMP. This provided information for analysis and enabled tasks to be traced throughout their life cycle. The set of tasks in DUMP contained a history of what had passed through the model. The function of the task expiry algorithm was to remove the expired tasks from the USER buffer and release the associated node:-

```
PROGRAM LISTING
Test time                          BEGIN TIMEUP
interval counter                   TIME OF TIMEUP LE 0
                                   CHAIN
Check that ACTIVE and              EXISTS NODE IN ACTIVE
USER are occupied                  AND EXISTS TASK IN USER
Find first                           FIND FIRST TASK J IN USER
expired task                         WITH TIME OF TASK LE 0
Transfer to DUMP                     TASK J FROM USER INTO DUMP
Release attached                     NODE (EXEC OF TASK J)
node                                 FROM ACTIVE INTO FREE
Reset time                           TIME OF TIMEUP = 1
interval counter                     REPEAT
```

5.11 The Reporting Process

Using the ECSL 'PRINT' statement it was possible to report on the entry of a task into a buffer and give various details about the state of the system. These were arranged to give

results in a tabular form in Section 6.4 (Tables 1 - 5).

It can easily be seen that for a low rate of input, the priority process has little meaning and very low priority tasks generated before higher priority tasks escape selection by priority when the buffer queue is empty.

5.12 Comment on the Model

The model was installed to provide a vehicle for the closer examination of semaphores and process synchronisation. The experiments carried out on the model are reported in Section 6. The discipline necessary to build a model that would compile and run ensured that no 'loose ends' were left unattended to. All buffers had to be properly managed and tasks selected for execution from the proper subset.

6 THE SIMULATION OF SOME CONCURRENT PROCESSING CONTROL MECHANISMS

Using the established model described in Section 5, control structures were added, to examine their effect on the model's behaviour. It was also considered important to determine if limitations were imposed on the control structures by the simulator. There were a number of basic control structures that could be provided. It was found possible to create an input stream that came from a fixed pool of processes. These were activated by a semaphore primitive (its simulated equivalent), as a counting semaphore that was incremented, to indicate that another execution of that process was required. Other structures involved the addition of further queues, into which active tasks had to be placed when they reached an occupied critical region or a synchronisation point. In the latter case, queuing became relevant if the anticipated synchronisation semaphore had not yet been received. In both cases queues were needed to avoid inefficiency and even deadlock. All available nodes might have become occupied by waiting tasks unless preventive measures were taken.

The implementation of these structures, using the basic simulation as a test bed, are described below, along with the reported results and conclusions drawn.

6.1 The Implementation of a Counting Semaphore

Using a set of inactive tasks, a counting semaphore was used, as an attribute of each task. When the semaphore was non-zero it made that task eligible for transfer to the active stream

(provided there was a vacancy in the buffer). While this resource was denied, the task had to remain in the inactive set, accumulating process calls on its counting semaphore. It was only necessary then to invoke the routine to increment this semaphore in order to make it a contender for the resource (the buffer). Any possibility of retaining exclusion on a particular node, after the task had been completed, in order to re-test the semaphore, was avoided in the model. Other task streams were thus allowed a possibility for entry. Each scan of the inactive set was independent of any other.

Following Keedy, Ramamohanaroo and Rosenberg [37] on the implementation of semaphores with sets, examination of the existing basic model reveals that the buffer NEXT was controlled by the counting variable B, which represented the number of available places (resources) in NEXT.

Applying a counting semaphore, as a condition for entry into buffer NEXT we have :-

```
PROGRAM STEPS                          P/V TERMINOLOGY

Task queue

    !
    v
Increment Semaphore                    signal (SEM)

    !
    v
  B < 5

    !
    v
Move task into NEXT

    !
    v
Increment B

    !
    v
Decrement Semaphore                    wait (SEM)

    !
    v
Task runs to completion

    !
    v
Decrement B
```

When implemented in the model it became obvious that only the subset with SEM > 0 should be scanned for the highest priority. A further variable was demand, represented by the value of the semaphore itself. The absolute value of the semaphore was not included as a criteria in the model's task selection algorithm.

6.2 The Implementation of Task Synchronisation

In order to cause an event in the lifetime of a running task, an event was assumed at (RunTime/2) + 2. This allowed both Producers and Consumers to operate realistically within their lifetime. They would not typically coincide as their start and run times would almost certainly differ, being derived from a

pseudo-random sequence. Synchronisation was established around a single Boolean (binary) semaphore, an attribute of the Consumer set. Consumers, not finding their semaphore set at the time (RunTime/2) + 1, were suspended in a queue and their resources released for other tasks. Their replacement came preferably from the same queue of suspended tasks. This was necessary, to clear suspended tasks from the SUSPEND buffer. The 'clocks' of these tasks were reset on activation to RunTime/2 which allowed the task to be re-tested in the semaphore routine at the next system clock cycle.

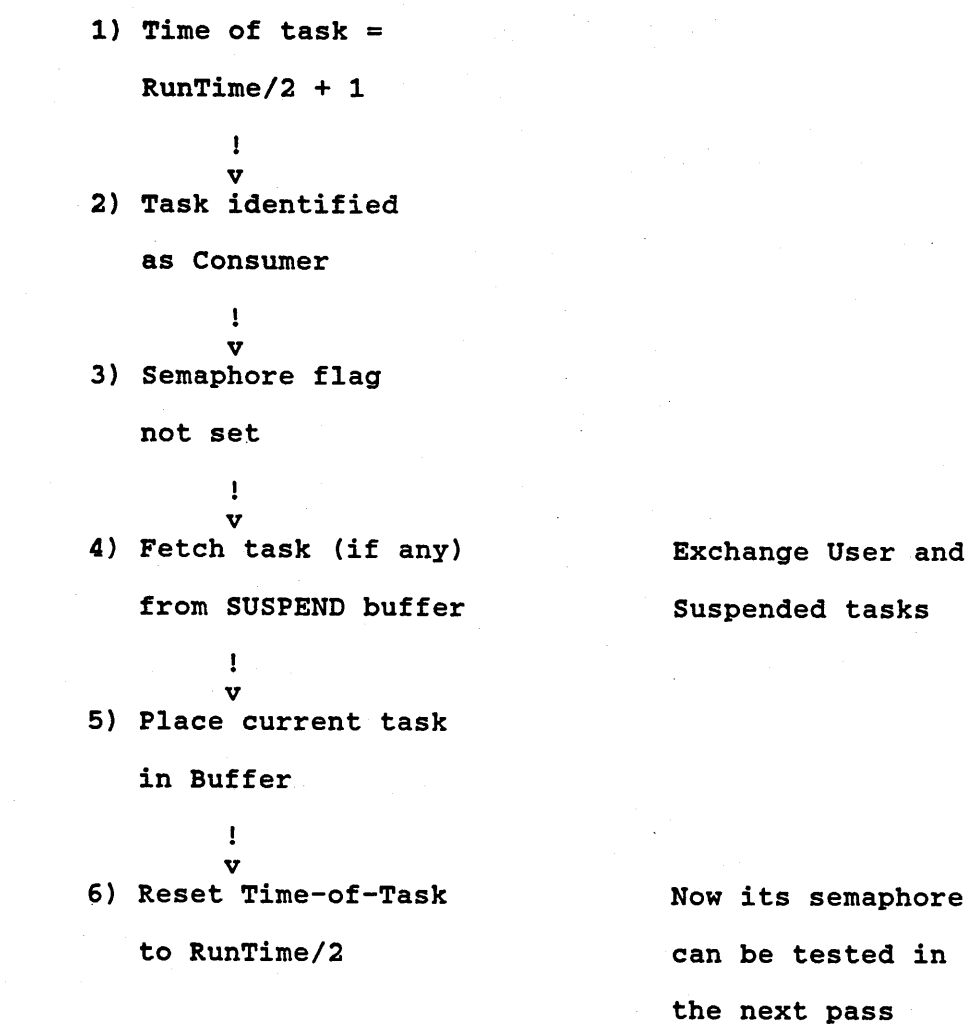
In order to control the semaphores, the most suitable structure found was a pair of sets to indicate:-

i) the Producers and their specified Consumers

ii) the Consumers and their synchronisation semaphores.

These two sets complemented each other and needed to be created from the information accompanying each task (in the model they were pre-set). This structure can be used for deadlock detection. Where the same group of tasks are present in both sets, there is the possibility of a deadlock.

Flow Chart I - Semaphore Reading



There were dangers in steps 4) and 5), a situation might have been reached where the retrieved task from the Suspend queue and that in User were both unsatisfied Consumers. The shorter the queue in suspension, the more likely this would have happened. Step 4) could have been omitted if a counter, incremented in 3), achieved too high a value and so indicated a need for new tasks to enter the system and break the impasse. The counter would then be set back to zero. No decision was made about the nature of the suspended task queue, whether it was a queue common to all nodes or specific to each.

Flow chart II - Semaphore Writing

1) Time of Task =

(Runtime / 2) + 2 Write after Read

!
v

2) Task identified

as Producer

!
v

3) Set semaphore

flag in allocated

Consumer

The results of implementing this algorithm are shown in Result Tables 4-5. In Table 4 the chain of events given below is simulated:-

Task1 -------> Task2 -------> Task3

For Table 5, the chain is closed, head to tail, to form a loop that cannot be completed and is in a state of deadlock:-

Task1 -------> Task2 -------> Task3
! !
<------------------------------

6.3 Producers and Consumers in Sets

The development of the synchronisation mechanism was directed at creating an environment where deadlocks and lock-outs could occur, and be detected. The creation of a cyclic structure can result in a deadlock, the minimum membership is two tasks. The deadlock may be endemic in the software (and detectable by

the compiler) or perhaps caused by a node failure. In the latter case, a re-try after the reorganisation of node activities may correct the situation.

If Producers and Consumers of a synchronisation process are recorded in sets it is possible to identify a possible state of deadlock, either by the cyclic nature of the entries or by the condition :

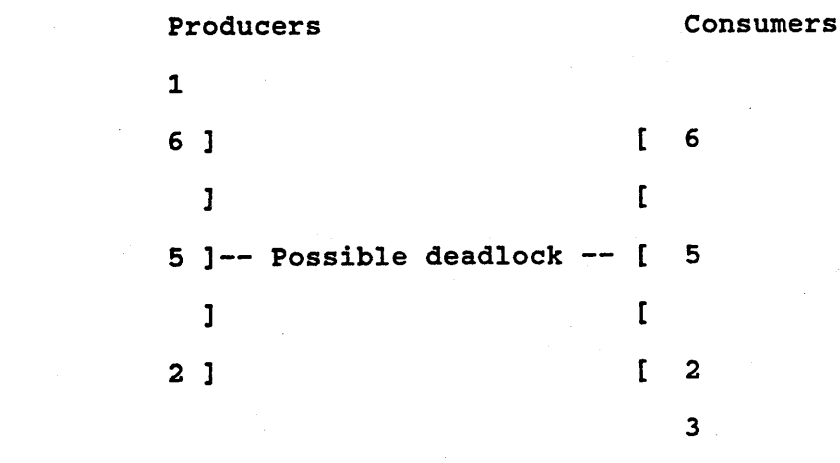
$$\text{Producer} \supseteq \text{Consumer}$$

remaining true for any period. This shows a dependence of a group of consumers on an overlapping group of producers, indicating a broken chain or a cycle of semaphores.

e.g.

1 --> 6 --> 5 --> 2 --> 3

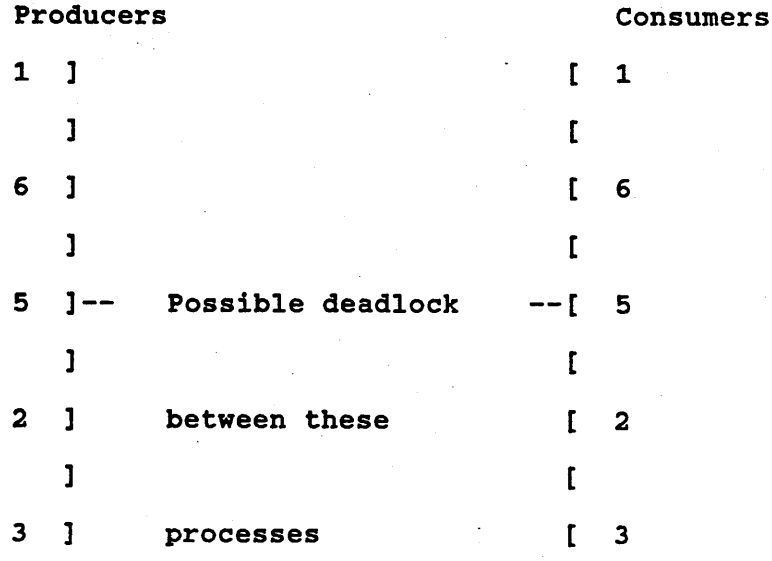
giving:-



If this becomes cyclic:-

1 --> 6 --> 5 --> 2 --> 3 --> 1

it gives:-


Producers
Consumers
1
6
5
2
3
1
6
5
2
3
Possible deadlock
between these
processes

6.4 Results Tables

Table 1

TASK IN BUFFER	NO HELD	TASK ACTIVE	ON NODE	RUN TIME	PRIORITY	TASK COMPLETED	NODE FREE
1	1						
		1	1	10	0		
						1	1
2	1						
		2	2	38	7		
3	1						
		3	3	51	8		
						2	2
						3	3
4	1						
		4	4	22	0		
						4	4
5	1						
		5	5	22	4		
6	1						
		6	1	10	0		
7	1						
		7	2	38	7		
8	1						
		8	3	51	8		

Explanation:-

At line 1, a task (No 1), entered the buffer. This task was allocated to Node 1 with a run time of 10, in line 2. Line 3 saw this task completed and the node freed. Subsequent tasks followed, being allocated to nodes in sequence. The arrival rate of the tasks was slow enough for all arriving tasks to be allocated to a node without a queue forming in the buffer (NO HELD = 1)

Table 2

- Waiting -		: ------ Executing ------				: -- Finished--:	
TASK IN BUFFER	NO HELD	TASK ACTIVE	ON NODE	RUN TIME	PRIORITY	TASK COMPLETED	NODE FREE
-	-	-	-	-	-	-	-
11	1						
12	2						
13	3						
						7	2
14	4						
		11	2	38	7		
						6	1
		12	1	10	0		
						8	3
		13	3	51	8		
						9	4
15	2						
		14	4	22	0		
						12	1
16	2						
17	3						
		15	1	10	0		
18	3						
19	4						
20	5						
						13	3

Comment:-

A higher arrival rate than Table 1 causes some queuing in the buffer. These results show the point where queuing began, at the arrival of task 12, earlier tasks having passed straight through in the manner of Table 1.

Table 3

TASK IN BUFFER	NO HELD	TASK ACTIVE	ON NODE	RUN TIME	PRIORITY	TASK COMPLETED	NODE FREE
-	-	-	-	-	-	-	-
8	1						
9	2						
		8	2	38	7		
						4	4
10	3						
		9	4	22	0		
12	3						
13	4						
14	5						
		6	3	51	8		
						5	5
15	5						
		10	5	22	4		

Comment :-

In this set of results, a fast arrival rate has caused early queuing. As soon as nodes are free, they are reallocated, both here and in Table 2, indicating a very efficient use of the nodes.

Table 4

TASK IN BUFFER	NO HELD	TASK ACTIVE	ON NODE	RUN TIME	PRIORITY	TASK COMPLETED	NODE FREE
1	1						
		1	1	10	0		
2	1						
		2	2	17	2		
3	1						
		3	3	56	2		

Comment:-

Tasks entered the USER buffer and continued to completion, despite a chain relationship between the tasks of 1-2-3. This is because the run-times of these associated tasks were in ascending order and they were executed in parallel. (1) will therefore have reached its Producing stage (at T10/2 +1) before (2) becomes a Consumer (at T17/2), and the same for (2) and (3).

Table 5

TASK IN BUFFER	NO HELD	TASK ACTIVE	ON NODE	RUN TIME	PRIORITY	TASK COMPLETED	NODE FREE
TASK 2 REACTIVATED							
	2		1	17	2		
	6		2	20	0		
TASK 2 SUSPENDED							

Comment:-

The full table of results began as Table 4 but there was now a cyclic relationship between 1-2-3, as 1-2-3-1. This resulted in the continued suspension and re-activation of all three tasks, which are never allowed to complete. The reactivated tasks were re-tried on the first available node. (1), (2) and (3) were members of both the Producer and Consumer sets.

6.5 Observations

A discrete event simulator was found to be a useful tool for examining concurrent processing algorithms. However there were aspects of its operation that could lead to misleading results. Routines simultaneously entering a section of critical code could not be directly modelled. The operation of the simulation software was essentially sequential within a given time interval scan and the effect of non-exclusion from a critical region not obvious when this aspect was considered for simulation. To properly represent a concurrent computer system, the effect of independent processors making simultaneous access attempts to resources, needed to be simulated. In a tightly-coupled system,there could be problems with access to multiport memories unless there is effective exclusion from critical regions for more than one processor. A simulator would need to properly represent events relating to the concurrent access of multi-port memory by several processors operating concurrently. Loosely coupled systems communicating through dedicated channels should not exhibit this problem and it ought to be possible to simulate such systems by any suitable language that provides the necessary concurrency and interprocess communications.

The use of sets for the control of resources and the detection of deadlocks amongst groups of co-operating tasks was very attractive but implied the presence of a master controller (which could be any node). Where the deadlock was created by the failure to run of a low priority task, a dynamic priority scheme would have allowed that task to automatically rise in priority and to be run and to break the deadlock.

7 THE MODELLING OF A SYSTOLIC ARRAY IN OCCAM

In order to examine the problem of simulating loosely coupled parallel processing architecture, the concurrent programming language Occam was applied to the algorithm of the convolution of a data array. The program simulated part of a 1D systolic array. This application was chosen because of its simplicity, and, as the results were predictable at every stage, the proving of the software (and later hardware) models was made more certain. The version of Occam referred to is proto-Occam and ran on the Sirius microcomputer. Experience showed that the levels of concurrency were limited in this programming environment, a stack overflow resulting if limits were exceeded.

The following sections introduce the language Occam and discuss aspects of communication and synchronisation that are important in this type of application. The algorithm used throughout is image convolution, which was introduced in Section 4.4 and is further described below.

7.1 Image Convolution

Various filters can be applied to a 2D array representing the pixels of an image. Both Pratt [32] and Rosenfield [33] describe various methods for image convolution. A kernel W is moved across an image array so that all positions are used in the computation. A simple noise cleaning convolution can be achieved using a vector W which is composed of positive values representing a low pass filter[34].

An array such as :-

1 1 1

1 2 1

1 1 1

with the results normalised to unitary weighting to avoid introducing a brightness bias, can be applied to an image. The results of applying such arrays to noisy images are shown in Appendix 1.

7.2 The Occam programming Language

The Occam language [7],[35] is derived directly from Hoare's CSP[17]. It was produced as the lowest level of language to be available to users on the Inmos Transputer. Development environments are available where Occam runs on processors other than the Transputer.

Occam language has several features that make it suitable for use in a concurrent processing environment. These are discussed below.

7.2.1 Language Primitives

In Occam there are three primitives from which all other processes are constructed:-

1) An input process, the symbol ? denoting input, for example;

```
chan1? v
```

The process waits at this command for data to arrive on channel chan1 and places the input in variable v.

2) An output process, the symbol ! denoting an output, for example:

```
data! x
```

The process sends the data contained in variable x to another concurrent process and then waits for it to be received. Channels act as rendezvous points and can therefore be used for process synchronisation. The channel is a single process to single process communication and can only be configured as a broadcast medium if an array of n channels are all utilised by n concurrent processes. Occam is more suited to asynchronous than synchronous systolic configurations, as interprocess communications take place on a point-to-point basis.

3) An assignment, for example:

```
x  := v    -- where x is assigned the value held in v
```

7.2.2 Concurrency

The use of indentation and the word PAR denotes that concurrency is present, e.g.:

```
PAR
  WHILE TRUE
    VAR x:
    SEQ
      chan1? x          -- These two sections of code
      chan2! x*x           run in parallel and
  WHILE TRUE                        |
    VAR y:                          |[chan2]
    SEQ                             v
      chan2? y          -- communicate using chan2
      chan3! y+1
```

A replicator can be used with PAR to replicate the process a number of times, e.g.:

```
PAR i = [0 FOR 5]  --5 copies, corresponding to i=0-4
  WHILE TRUE
    VAR x:
    SEQ
      data[i]? x --input from data[0]....data[4]
      data[i+1]! x  -- output x to data[1]....data[5]
```

Which creates a 5-stage FIFO buffer whose input is data[0] and output is data[5].

7.3 Implementing Delay Structures in Occam

The delay process is shown later to be critical to the performance of a 1D arrangement of processing elements applied to nD convolution. A single cycle delay is needed at each computation stage, and longer delays at specific intervals. The concept of a cycle in what is essentially an asynchronously communicating environment can be applied to the passage of data, one item of data being passed equates to a cycle.

In Occam the simplest construct that could be used in a delay pipe-line is:-

```
SEQ
  in? x
  out!x
```

This has the one major disadvantage that it relies on a given sequence of events. The process can only deal either with input or with output, if these do not occur in the correct order,

a deadlock is created. A better arrangement is a double buffer routine which uses parallel constructs to handle input and output concurrently.

The order of in? and out! (passing data between parallel processes on Occam channels), is no longer significant. It is sufficient for them both to occur eventually:-

```
SEQ
  in? y                    --initial value in y
  WHILE TRUE
  SEQ
    PAR                    --parallel execution
      in? x                --next input in x
      out! y               --value in y output
    PAR
      out! x               --previous value in x output
      in? y                --new y input
```

A single cycle delay is not produced by either of the above routines unless initial action is taken to write data into the delay routine. Essentially, if the double buffer is empty, a process will first input a data item into y and then concurrently read a new item into x while outputting y. The input of data is always accompanied by the output of the previous data item. An initialisation of the buffer can be performed, writing a single item into y so that the normal operating regions of the double buffer process are in the concurrent regions:-

```
PAR
   in? x
   out! y
PAR
  out! x
  in? y
```

Without the initialisation data simply passes through the routine without any delay. Fig 7.2 illustrates this:-

Fig 7.2.

a)

```
delay!......> [y] ......> delay?
                [x]
```

Data passes into [y] and directly out again giving no delay

b)

```
          0.........>[y].......> data?
     Priming data          .
                      {next  cycle}
                           .
          delay!...>[x]
```

Priming data is written into [y]. Actual data enters [x] and is not written until the next cycle - next data sent

The additional input into the buffer is a once only event, a priming operation, from then on the output is one data cycle behind the input. This needs to be applied to all delays in the system.

7.4 n-Cycle Delays

Parallel structures, containing single cycle delays can be created - Fig 7.3.

```
----->[ D/B ]----->[ D/B ]----->[ D/B ]----->[ D/B ]----->
```

D/B = a single double buffer

This raises the interesting possibility that a partially initialised buffer chain will only delay the data stream by the amount that it has been primed. It is clear that i cycles of delay can be produced from a pipeline of m parallel double buffers if i items of data are entered prior to any data leaving the pipeline, where:-

$$0 <= i <= m$$

7.5 nD Convolution using Systolic Arrays

Using pipeline delays and initialising the appropriate delays with suitable values it is possible to change the nature of the process being undertaken. A 1D convolution array can be re-initialised to nD by selecting the appropriate delay pipelines and initialising them to produce the required delays. Kung[21] has described 1D systolic array capable of nD convolution, using delays where appropriate.

An Occam program could be set to carry out 1D or 2D convolution, according to a single logical control which would determine the extent to which the delays are initialised. For 1D convolution the two arrays must first meet as shown in Fig 7.4 a). The weight array [W] then moves along the data array [P] as shown in Fig 7.4 b)-c).

Fig 7.4.

a)

```
P1 P2 P3 P4 P5 P6 P7 P8 P9 P10 P11
W1 W2 W3 W4 W5 W6 W7 W8 W9
```

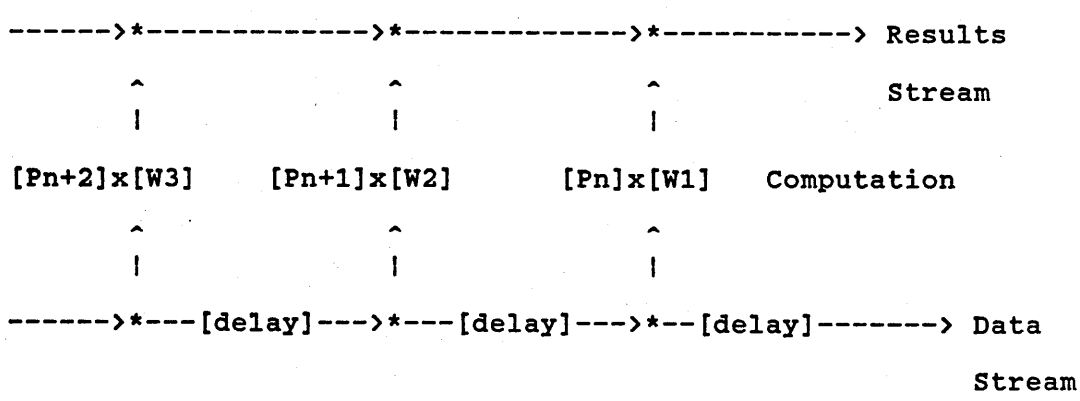
b)

```
P1 P2 P3 P4 P5 P6 P7 P8 P9 P10 P11
   W1 W2 W3 W4 W5 W6 W7 W8 W9
```

c)

```
P1 P2 P3 P4 P5 P6 P7 P8 P9 P10 P11
      W1 W2 W3 W4 W5 W6 W7 W8  W9
```

Where the array W is moved across an array P, the inner product calculation is made at every conflux.
The relationship between the data, results and delays is illustrated by:-



The single cycle delay has the required effect of moving one array over another.

For 2D image convolution, certain delays must accommodate the remaining data in a single line of the image (+1 cycle).

Fig 7.5

a)

```
P1    P2    P3    P4    P5    P6    P7    .  .  Pn
W1    W2    W3
Pn+1  Pn+2  Pn+3  .     .     .     .     .  .  P2n
W4    W5    W6
P2n+1 P2n+2 P2n+3 .     .     .
W7    W8    W9
```

b)

```
P1    P2    P3    P4    P5    P6    P7    .  .  Pn
      W1    W2    W3
Pn+1  Pn+2  Pn+3  Pn+4  .     .     .     .  .  P2n
      W4    W5    W6
P2n+1 P2n+2 P2n+3 P2n+4 .     .     .
      W7    W8    W9
```

Assuming an i x i filter array W applied to an n pixel wide image, the requirement is for the (ixj)th delay element to have a delay depth of (n-i)+1, where j= 1,2 ...i-1. In the context of the flexible delay pipelines, this is the extent of the priming required in order to delay the data not currently lying under the filter array. As the filter moves across the image array, the delayed data leaves the delay pipe and coincides with the appropriate element of W.

7.6 Process Synchronisation

In Occam no global synchronisation mechanism is provided. A channel input can be used to guard a process :-

```
CLOCK ? ANY        -- Guard Process
DATA-IN? x
```

This could be expanded to encompass a number of processes, using an array of channels :-

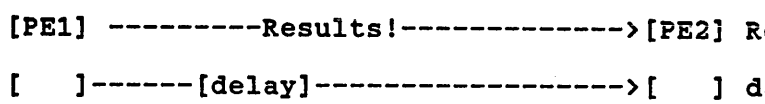
```
WHILE TRUE
  PAR  i = [0 FOR n]
    CLOCK[i]! n

     -
     -
     -
    CLOCK[i]? ANY
    DATA-IN? x
```

This attempt to impose a global process synchronisation on the Occam formalism could well result in deadlock or in processes getting one or more 'clock cycles' out of phase.

The flow of data and results through the 1D systolic array can be quite effectively synchronised by the results stream. Data will not 'back-up' in the delay pipe line as the new result cannot be transferred between processing elements until the previous result and the data item that it effectively guards, have been read:-

```
SEQ
  results? a              --from previous process
  data-in? b              --from delay pipeline
```

This can be represented as :-



A PE can make an output to the results channel only if the previous output has been read, there is no buffering in the Occam channels. This output is followed by data output to the delay channel.

7.7 The Occam Model

In Occam applications, concurrency can occur within a single processor system or across a group of processors. In the model, a single PE containing three sequential cells was simulated. Concurrency occurred within each sequential cell and related delay structures to an extent which prevented the cells being implemented in parallel, due to the limited stack space in the development environment. The three sequential cells did however fit well into the subsequent hardware array.

The Occam model is listed below :-

```
PAR
  WHILE TRUE
     DEF WEIGHT = TABLE [0001,0002,0001]: --Weight Array
     SEQ
       VAR R,D:  --Local variables
       SEQ  i = [0 FOR 3] -- Three sequential Cells,[0,1,2
         SEQ
           PAR
             RESIN[i]? R --Communicate between Cells
             DATAIN[i]? D
           R:=R + (D * WEIGHT[i]) --Computation
           PAR
             RESOUT[i]! R --Results stream
             DELAY[i]! D --Data into delay path
```

This process, when run in a suitable test harness, (Appendix 1) was used to model a single PE in the hardware design. Parallel processes were needed to provide Cell-Cell communication, data I/O and a delay path. These represent a considerable overhead on the basic process. A program to represent the full array would exhibit excessive concurrency. This would include the process as shown together with its buffering, in three concurrent copies, as well as the line delay consisting of 253 concurrent double buffer processes. The complete hardware design cannot be modelled in the available Occam programming environment without re-designing the algorithm or providing sufficient hardware for the concurrency required. However the results from a single PE modelled in Occam are predictable and provided valuable support for the development and practical implementation of a systolic array.

The results from a single PE show the effect of the Weight array moving across the Data:-

```
                                        [Weight Array]
a) Data Stream : 11111111111111111  * 121
   Results     : 13444444444444444
Produced by:-    121000000000000
                  12100000000000  +
                   1210000000000  + etc.
                = 13444.....

b) Data Stream : 111101111  * 121
   Results     : 134432344

c) Data Stream : 001000  * 121
   Results     : 001210
```

8 A SYSTOLIC ARRAY FOR IMAGE CONVOLUTION

In order to verify the mechanisms modelled in Occam, an asynchronous systolic array was designed, built and tested. It was then applied to the convolution of an image on to which noise had been injected. This section covers the key points in the design and verification process and prints of images are given in the text and in Appendix 1, before and after processing. The hardware was designed around three Intel 8748 single-chip microcomputers arranged in a 1D array. Parallel communication links were arranged between processors and to the data source/destination, implemented in software embedded in the 8748s.

8.1 The Processing Element

A single chip microcomputer was chosen as the basic array cell. Any hardware features not available in the device were to be provided in software and as such, could be easily altered. Any losses in performance resulting from implementing functions in software were discounted as performance was not a design objective. The Intel 8748 microcomputer used can function as a complete self contained computer, with 1K of user programmable/UV Erasable ROM, 64 bytes of RAM and 27 I/O lines. Communication between processors had to be carried out over the two 8-bit general purpose I/O ports, using a software controlled handshaking protocol.

The use of 8748 processors could in no sense be considered efficient in this application. Their purpose was to represent and carry out the functions of the cells designed and tested in Occam. Each processor carried out three sequential multiply-

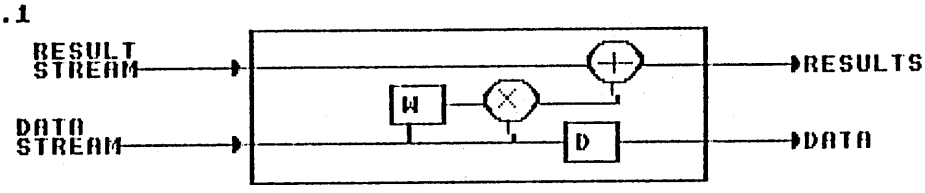
accumulate (convolution) computations (Fig 8.1), representing three systolic cells (Fig 8.2), and provided all the necessary delay paths for the convolution of a 2D data array by a 1D array processing elements (Fig 8.3). Although a common cpu clock was used, the processors operated asynchronously, the only synchronisation between them occurred during inter-processor communication, when the programs in adjacent processors had to rendezvous. The Weight array was broadcast to the individual cells when the run was initialised (Fig 8.4).

8.2 The Design of the Array Hardware

Each 8748 required two 4-bit results ports (in and out) and two data ports (also both in and out). These were arranged so that the input data and results were read at the same time, using a pair of handshake lines. A similar arrangement was made for output data, giving:-

```
4-bit result       ]
1-bit data         ] = 7 Port lines each for
2-bit handshaking  ]   input and output
```

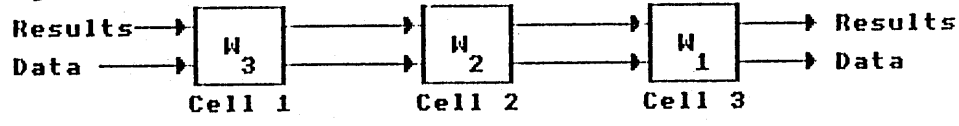
Fig 8.1



W = WEIGHT
D = DELAY
X = MULTIPLY
+ = ADDITION

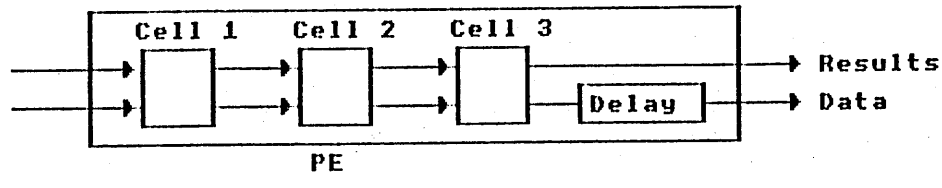
The computation performed in each cell.

Fig 8.2



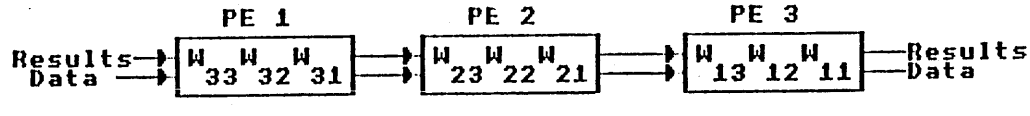
The relationship between the computational cells.

Fig 8.3



The cells in a single Processing Element (PE) with the associated delay.

Fig 8.4



The relationship between the three PEs showing the position of the weight values in each cell.

The eighth line (in each of the two 8-bit ports utilised) was used as a signal to indicate the broadcasting of weight values from the microcomputer used to control the array. Fig. 8.5 Port 1 of each 8748 was configured for the input of 7-bits of data, results and control signals, a single output line provided the handshake reply. Fig. 8.6 Similarly Port 2 was configured for 7-bits of output and a single input. Fig 8.7

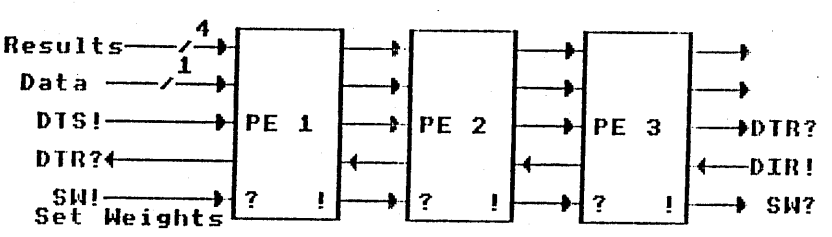
Apart from the three 8748s utilised in the array, very few other components were required to provide common clock and reset signals. All communication and delay functions were implemented in software. The array operated under the control of a dedicated microcomputer which both generated and received the image data to be processed.

8.3 Data Generation/Reception

A BBC Model B microcomputer was used to control the array. A fixed test image was used, to which was added a variable (selectable) amount of random noise. The image was displayed (BBC-Mode 0) and communicated as 1-bit pixels. Pixels were simultaneously output from the array and replaced on the screen image, as they became available from the array.

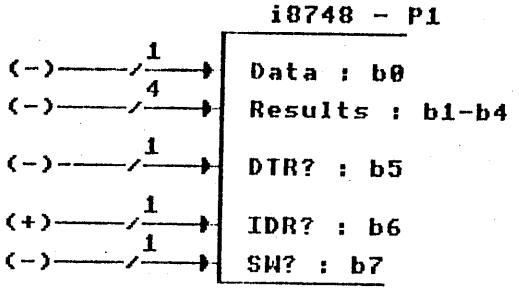
The image was dimensioned as 256 x 256 pixels and output from the BBC as a series single bit pixels This was transmitted over a parallel interface compatible with the processor-to-processor communications implemented between each 8748. The individual processors contained identical software. In the BBC computer the 4-bit result leaving the array was normalised by the sum of the weights used, to produce a one bit image pixel.

Fig 8.5



The Interface between the Array and the BBC Micro

Fig 8.6

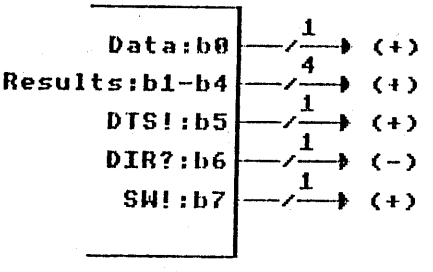


(+) = To next PE

(-) = To previous PE

Input I/O in each PE

Fig 8.7



Output I/O in each PE

Menu driven software in the BBC computer allowed the selection of a specified number of randomly placed black or white noise pixels which were added to the fixed image of a cross. Other options provided image transmission/reception and the broadcasting of a new weight array to each of the nine cells in the array. This latter operation needed to be accompanied by a general reset of the array and utilised the SW signal implemented on the processor-to-processor interface.

8.4 Implementation of the Hardware

In order to examine the operation of a systolic array before the implementation of the hardware a cell of the array was modelled in Occam and its characteristics examined (Section 7.7). These were subsequently compared with the characteristics of an actual cell and a close relationship was found. The need to fill and flush internal buffers was a common feature in both cases. The priming data needed to be entered into the delay pipelines, before true results were output. At the end of the run, results remained in the internal buffers which had to be flushed out by inputting null data values. An identical (and predictable) results stream was produced by the same test data when applied to both systems.

Aspects of the design and application of this hardware are further discussed below.

Fig 8.8 The Software Process In Each PE.

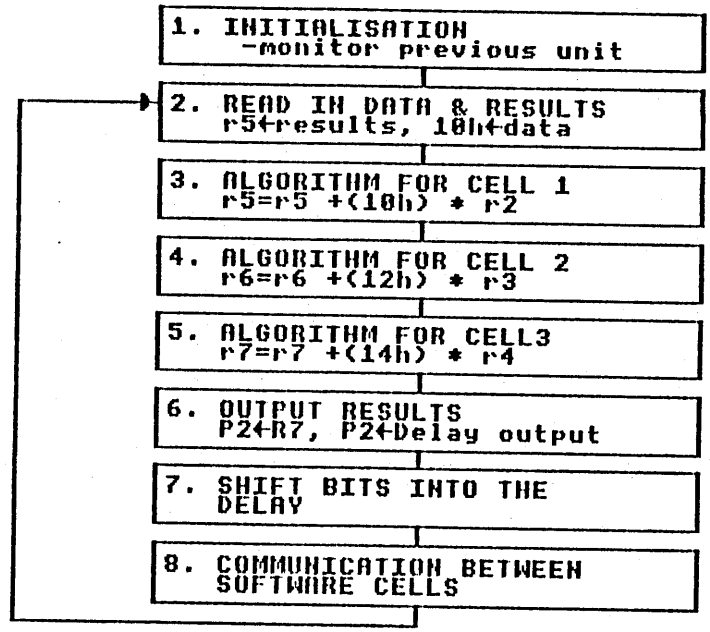
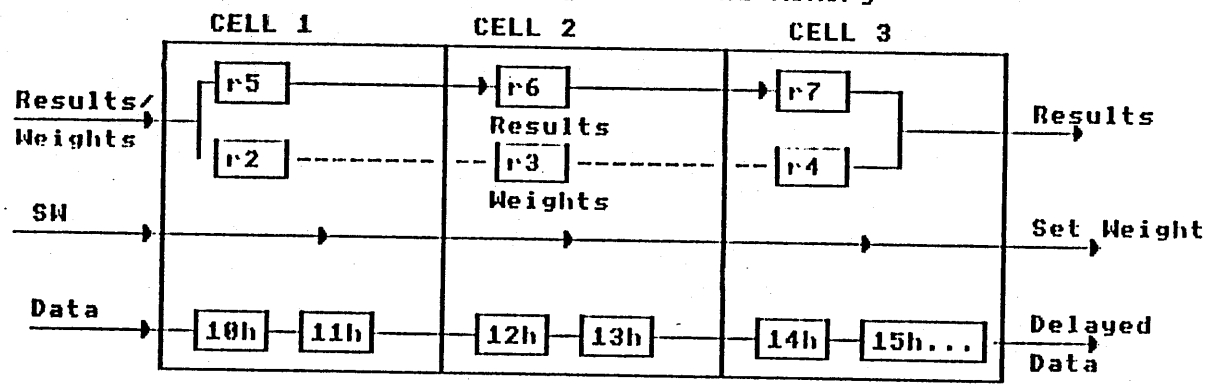


Fig 8.9 The Use Made of i8748 Registers and Memory



8.4.1 PE-PE Communication

The Data path(channel) used to communicate between processors was one bit wide across the array. The Results path was limited to four bits wide, due to I/O limitations. The 4-bit results were produced from adding the partial results from each cell into the results stream. These partial results were derived from the product of the 1-bit data stream and the stored(broadcast) weights.

By limiting the maximum sum of the weights used to 15, the maximum result value was contained within the 4-bits limit of the results interface, for example the weight array:-

1	1	1
1	2	1
1	1	1

has the maximum result value limited to 10.

The data transfer rate between processors was slow, limited by the software processes having to deal with the convolution algorithm, the delay processes and I/O. Fig 8.7. A rate of of 550 baud was measured, corresponding to a calculated cycle time for each PE of 1.88 ms (532 baud).

8.4.2 Cell-Cell Communication

The sequential software process in each PE involved the operations shown in Fig 8.8, utilising the 8748 register-file shown in Fig 8.9. Each cell computed a result, e.g.

$$r5 = r5 + [(10h) * r2]$$

Where memory location 10h(hexadecimal) contained the input data and r2 the relevant weight value.

Once I/O processes and delay operations for the data stream were completed, r6 was copied into r7 and r5 into r6, ready for a new result value to enter r5. A similar process was applied to the data variables at addresses 10h,12h and 14h, which passed through the delay process described in 8.4.3.

8.4.3 The Delay Processes

Each cell utilised even numbered memory locations for data variables. the following odd numbered locations were used to implement a one-cycle delay(*) by shifting data up one place each cycle:-

```
                        *         *         *
[INPUT]-->10h-->11h-->12h-->13H-->14h-->15h-->[OUTPUT]
```

The delaying of the data by the difference of the width of the image and the width of the weight array, was achieved by implementing a shift algorithm which created a 'shift register' of the required length in memory. Each memory byte was rotated once, the carry being used to link the operations. The number of rotate operations was computed by:-

(delay length/8) + 1

8.4.4 Broadcasting the Weight Values

A method was provided to allow the broadcasting of weight values to the individual cells. The 4-bit results channel was used, during a period of intiallisation while the SW signal was active, to broadcast new weights. The weight values were limited to the width of the results channel.

Fig 8.10 The First Image Print

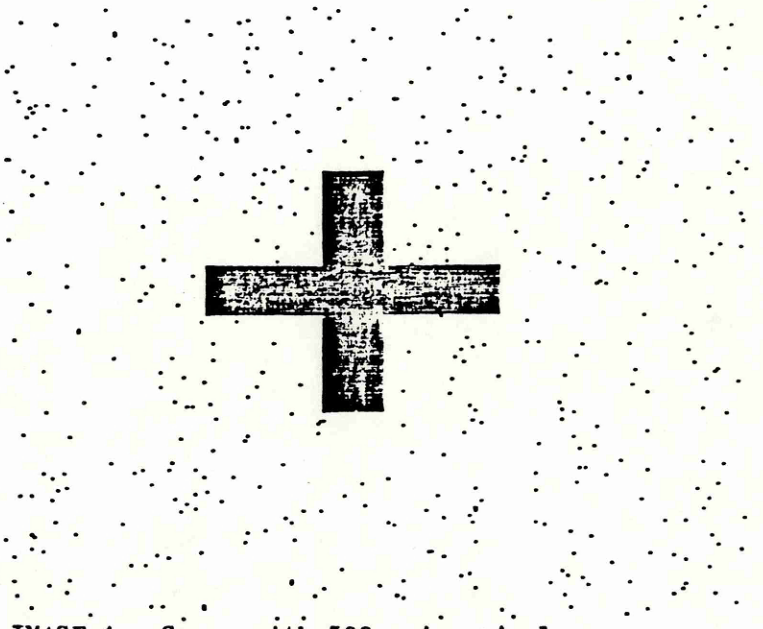


IMAGE 1. Cross with 500 noise pixels

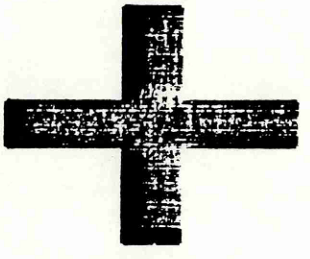


IMAGE 2. Result of IMAGE 1 being processed by a 1 1 1 kernel.
1 2 1
1 1 1

8.5 Processing the Image

The image data was sent through the array by fetching the first pixel of the first line and then scanning the image line by line. Valid results did not emerge from the array until all the delay processes were completed. Once the array had been filled with data and the third pixel of the third line sent, the generated result emerging was valid:-

line1 - p1 p2 p3 p4 .....p256

line2 - p257[p258]p259 p260.....p512

line3 - p513 p514 p515 p516.....

[px] = the first pixel sent to be at the centre of a 3*3 array of pixels. The pixels and their relevant weight array locations for the first valid computation are:-

p1*w1 p2*w2 p3*w3

p257*w4 p258*w5 p259*w6

p513*w7 p514*w8 p515*w9

Valid data was written back into the image array from pixel 258 onwards. Once the whole image had been transmitted some results remained in the array which needed flushing out using zero value inputs for results and data. The effect of the initial invalid results and results left behind in the array can be seen in the image print Fig. 8.10 and in Appendix 2. Edge effects were accounted for by ignoring the two invalid results at the end of each line as the weight array no longer overlays related image pixels.

8.6 Efficient Operation of the Array

The input of data into the array needed to be of a sufficient rate to satisfy all demands for data made by the processing elements, if the array was to operate efficiently. Due to the sequential nature of each PE, communication between them could only take place as shown below:-

[BBC OUTPUT]---->PE1, PE2->PE3

PE1->PE2, PE3---->[BBC INPUT]

[BBC OUTPUT]---->PE1, PE2->PE3

etc.

A low input rate would result in cells waiting for data.

8.7 Synchronisation

Individual processors were driven, for convenience, from a common clock. Apart from this they operated asynchronously and data communication was under local and not global control, matching the synchronisation found earlier in Occam channels.

8.8 Results

A sample of the images processed was given in Fig 8.10 and further images are to be found in Appendix 1. Various filters (weight arrays) were applied, all with positive integers as negative values were not accommodated in the design. The results shown were predictable and verify that the architecture of the processor array was correct and that all the delay, I/O and arithmetic processes were functioning properly.

Conclusions relating to these results are given at the end of this thesis.

PART 4

9 CONCLUSIONS

The examination of parallel processing mechanisms and architectures by simulation reveals a great deal about their apparent behavioural characteristics. There is always a possibility that the simulation process itself distorts the subject because of its own limitations. As a result there may not be a true relationship between the model and the actual system under investigation, leading to incorrect results.

The emulation of the system, using a combination of hardware and software components, can help to confirm the integrity of a design that has already been successfully simulated. Problems relating to the asynchronous nature of independent processors may be hidden by the operation of the simulator. Despite the limitations in performance of a hardware model made up of off-the-shelf components that are chosen because of availability rather than high bandwidth, the exercise can be a useful and rewarding one. Within programmable components in the model, missing hardware features such as communication channels, can be replaced by a combination of hardware and software, to implement handshaking over a parallel I/O port.

The simulation of concurrent processes, interacting and communicating with each other, shows many areas where the design may need attention. Both the length and membership of queues of processes are aspects that can be altered at short notice and the effect of any changes observed.

The clock of the simulator dominates the model. It is assumed that concurrent events happen in synchronisation with a

common clock. Before a simulation is attempted it is necessary to decide what aspects in the system are to be the subject of the simulation and what design features are predetermined.

It is possible to construct a model of a parallel computer system using a concurrent programming language such as Occam. By defining which parallel routines represent code in each of the simulated processors, it is possible to represent the entire system in code that is executed on a single processor. Communication between processes is important and a method such as the Occam channel allows inter-process communication and synchronisation to remain the same irrespective of the number of processors on which the algorithm is running.

The comparison of an implementation of a concurrent algorithm in Occam on a single processor system, with the emulation of the same algorithm in a multi-processor environment showed a close relationship. The concurrency expressed in the algorithm produced the same results irrespective of the hardware configuration. Communication and synchronisation through channels, allowed concurrent processes to be implemented and tested in the development environment. They were then re-implemented in a hardware model that matched the parallelism expressed in the algorithm, in order to verify the original model.

The results indicated that the Occam channel is a powerful construct, allowing parallel algorithms to be run on a variety of architectures without changes being made to the code.

10 FUTURE RESEARCH

Multiprocessor architectures are becoming increasingly important and the building blocks, in the form of VLSI components, are now available for the construction of new designs. Dataflow processors are now available and the design of a multiprocessor dataflow image processing system, using advanced simulation packages (ELLA, HELIX etc.) to support the design process, is the proposed next step in this research. The finished product will utilise vlsi dataflow processors [53] operating under the control of a development system. This would be used with a range of image processing algorithms, with a view to its use with X-Ray picture enhancement and analysis.

SOURCES

Information sources for the preparation of this thesis have included the British Library at Boston Spa, Leeds Polytechnic Library, Leeds University Library, the Oxford University Programming Research Group and the British Computer Society Journal.

The references included overleaf are the major ones that have influenced the course of this research. I am grateful to all concerned for taking the time and trouble to provide me with this valuable material.

REFERENCES

1. Thornton, J.E., 'Parallel Operation in the Control Data 6600' AFIPS Conf. Proc. 26 (II) 33-40.

2. Connors, W.D. Florkowski, J.H., and Patton, S.K., 'The IBM 3033: An Inside Look' Datamation, May 1979, pp 198-218.

3. Flynn, M.J. 'Some Computer Organisations and their Effectiveness' IEEE Trans. Comput. C-21 pp 949-960.

4. Feng, T.Y., 'Some Characteristic of Associative Parallel Processing' Proc Sagamore Comp. Conf. 1972 pp5-16.

5. Shore, J.E., 'Second Thoughts on Parallel Processing' Comput. Elect. Eng 1. pp 95-109

6. Kogge, P.M., 'The Architecture of Pipelined Computers' McGraw-Hill 1981 ISBN 0-07-035237-2.

7. Cray Research Inc., 'CRAY-1 Computer System' No 2240004, Cray Research Inc., Bloomington, Minn.

8. Reddaway S.F., 'DAP - A Distributed Array Processor' 1st Ann. Symp. on Comp. Arch. (IEEE/ACM).

9. Higbie L.C., 'The Omen Computers: Associative Array Processors' COMPCON Digest pp 287-290, 1972.

10. Berg, R.O., Schmitz, H.G., & Nuspl, S.J. 'PEPE- an overview Architecture, Operation and Inplementation' IEEE Nat. Electron. Conf. 1972 : 27 pp 312-317.

11. Slotnik, D.L., 'Unconventional Systems' AFIPS Conf. Proc. 1967 : 30 pp477-481.

12. Fuchs, H. et al, 'Fast Spheres, Shadows, Textures, Transparencies and Image Enhancements in Pixel-Planes' Proc. Siggraph '84 Computer Graphics v.19 no.3, pp111-120.

13. Handler, W., 'The Impact of Clasification Schemes on Computer Architecture' Proc. 1977 Int. Conf. on Parallel Processing, pp 7-15.

14. Watson, W.J., 'The Texas Instruments Advanced Scientific Computer' COMPCON 1972 pp 291-294.

15. Burroughs Corp., 'Burroughs Scientific Processor - overview, perspective & architecture', Burroughs Doc. 61391A.

16. Vick, C.R. & Cornell, J.A., 'PEPE Architecture - present and future', AFIPS Conf. Proc. 47 pp981-1002, 1978.

17. Batcher, K.E., 'The STARAN Computer' Infotech State of the Art Report: Supercomputers, Infotech 1979, pp 33-49.

18. Goodyear Aerospace Co., 'Massively Parallel Processor (MPP)', Tech. Report GER-16684, July 1979.

19. Feng, T.Y., 'A Survey of Interconnection Networks' IEEE Comp., Dec 1981, pp 12-27.

20. Bashkow, T.R., 'Supercomputers: always in the fast lane', Comp. Tech. Rev., Vol VII No12. OCT. 1987.

21. Fischer, W., 'The VMEbus Project', Proc Ieee Compcon, Feb 1984 pp376-378.

22. Dijkstra, E.W., 'Cooperating Sequential Processes' - Programming Languages, pp 43-122 Academic Press 1968.

23. Reyling, G.R., 'Performance and Control of Multiple Microprocessor Systems' Computer Design Mar. 1974 pp81-86.

24. Keedy, J.L., Rosenberg, J. & Ramamohanarao, K., 'On Synchronising Readers and Writers with Semaphores' Computer Journal Vol. 25, No. 1., 1982 pp 121-125.

25. Pyle, I.C. 'The Ada Programming Language' Prentice-Hall 1981

26. Inmos, 'The Occam Programming Manual'.

27. Ben-Ari, M., 'Principles of Concurrent Programming' Prentice-Hall 1982.

28. Keedy, J.L., 'An Outline of the 2900 Series System Architecture', Australian Computer Journal 9 (No.2) pp53-62

29. Motorola 'The MC68000 Programming Manual'.

30. Inmos, 'IMS T424 Transputer User Manual'.

31. Hansen, P.B., 'The Programming Language Concurrent Pascal' IEEE Transactions on Software Engineering Vol SE-1 No 2 June 1975

32. Hoare, C.A.R., 'Monitors: An Operating System Concept' Communications of the ACM Oct 1974 Vol 17 No 18.

33. Hoare, C.A.R., 'Communicating Sequential Processes' Communications of the ACM Aug 1978 Vol 21 No 8.

34. Kuo S.S., Linck M.H., Saadat, S., 'A Guide to Communicating Sequential Processes' Oxford Univ Monograph PRG-14.

35. Hansen, P.B., 'Distributed Processor - A Concurrent Programming Concept', Comm ACM Vol. 21 No. 11 pp934-941.

36. Users Manual 'Extended Control and Simulation Language' University of Birmingham.

37. Kung, H.T., & Leiserson, C.E., 'Algorithms for VLSI Processor Arrays' Section 8.3 - Mead and Conway.

38. Kung, H.T., & Pickard, R.L. 'Hardware Pipelines for Multidimensional Convolution and Resampling' Proc. 1981 IEEE Computer Society Workshop Nov 1981 pp273-278

39. Kung, H.T., 'Why Systolic Archtectures' Computer Jan 1982.

40. Kung, H.T., Ruane, L.M. & Yen, D.W.L., 'Two-Level Pipelined Systolic Array for Multidimensional Convolution' Image and Vision Computing Vol 1 No 1 Feb 1983.

41. Kung, S.Y., 'On Supercomputing with Systolic/Wavefront Array Processors' Proc IEEE Vol. 72. No. 7.

42. McCanny, J.V. & McWhirter, J.G., 'Implementation of Signal Processing Functions using 1-bit Systolic Arrays' Electronics Letters. 18 No. 6 pp241-243 March 1982.

43. Corry A. & Patel, K. 'Architecture of a CMOS Correlator' IEEE 1983 Int Symp. on Circuits and Systems May 1983.

44. Hooper D., 'Aspects of Processor Architectures' University College, London 1983.

45. Arvind, D.K., Robinson, I.N. & Parker I., 'A VLSI Chip for Real-Time Image Processing' roc. IEEE Symp. Circuits & Systems May 1983.

46. NCR 'GAPP Product Description'.

47. Dew, P.M., Dodsworth, J. & Morris, D.T., 'Systolic Array Architectures for High Performance CAD/CAM Workstations' Report no. 201 University of Leeds, Dept of Computer Studies June 1985.

48. Barron, I.M. 'The Transputer' Proceedings - Westcon MiniMicro 1983.

49. Pratt, W.K. 'Digital Image Processing' John Wiley 1982.

50. Rosenfeld, A. 'Picture Processing by Computer' Academic Press 1969.

51. Hilditch, C.J., 'Comparison of Thinning Algorithms on a Parallel Processor' Image and Vision Computing Vol 1 No 3 Aug 1983.

52. May, D. & Taylor, R. 'Occam - An Overview' Microprocessors and Microsystems Vol 18 No 2 March 1984.

53. NEC Electronics (Europe) Gmbh Product Description uPD7281 Image Pipelined Processor - Application Library 1986.

APPENDIX 1

OCCAM LISTING

The program listed overleaf consists of the routine containing three sequential cells, listed earlier, set in a test bed to provide screen and keyboard I/O.

```
-- Test Harness for Single PE Model (3 Cells)  --


-- This program runs the three sequential cells, communicating
-- by channels in processes running in parallel. Data and a zero
-- results value are entered by a parallel process
-- into Cell[0]. Results from the PE, output from Cell[2]
-- are picked up by another parallel process, converted
-- to ASCII and displayed. The output data channel is also
-- read, to prevent deadlock.


CHAN RESOUT[3],RESIN[3],DATAIN[3],DELAY[3]: --Channels used in PE
SEQ
  WHILE TRUE
    PAR
      WHILE TRUE
       DEF WEIGHT = TABLE[ #0001, #0002, #0001]: --Weights in this PE
       SEQ
           VAR R,D:
           SEQ i = [0 for 3]  --Three Sequential Cells
             SEQ
               PAR
                 RESIN[i]? R   -- Results input
                 DATAIN[i]? D  --Data input
               R:= R + (D * WEIGHT[i]) --Computation
               PAR
                 RESOUT[i]! R  --Results output
                 DELAY[i]!  D  --Data into delay
      WHILE TRUE
        PAR J = [0 FOR 2]
          VAR x:
          SEQ
            RESOUT[J]? x   --Transfer results between cells
            RESIN[J+1]! x  --without any delay
      WHILE TRUE
        PAR k = [0 FOR 2]
          VAR x,y:
          SEQ
            DELAY[k]? y  --First delay input (primins)
            WHILE TRUE
              SEQ
                PAR
                  DELAY[k]? x  --Double buffered
                  DATAIN[k+1]! y --delay routine
                PAR
                  DATAIN[k+1]! x --Return data to cell
                  DELAY[k]? y    --Read more data
       WHILE TRUE
           VAR d,e:
           PAR
             SEQ
               d:=0
               RESIN[0]! d --Enter 0 value in Results[0]
             SEQ
               KEYBOARD? e --Keyboard entry of data
               e:= e-'0'   --Reduce ASCII to Binary
               DATAIN[0]! e --Input to 1st cell
       WHILE TRUE
           VAR f:
           SEQ
             PAR
               RESOUT[2]? f --Pick up results from PE
               DELAY[2]? ANY  --Read data to prevent deadlock
             SCREEN! (f+'0') --Convert Binary to ASCII & Display
       SEQ
         VAR T:
         SEQ
           T:= 0           --Priming routine for Delay
           PAR             --A 'once only' operation
             DELAY[0]! T  --Delay[0] primed
             DELAY[1]! T  --Delay[1] primed
```

APPENDIX 2

IMAGE PRINTS

The results shown below show the effect of filtering test images with various levels of noise present. In some cases the rounding of sharp edges can be observed. Where the applied random noise was of a high density, the noise itself can be seen to have formed occasional regions that are so dense that they cannot be filtered out with the limited filtering available.

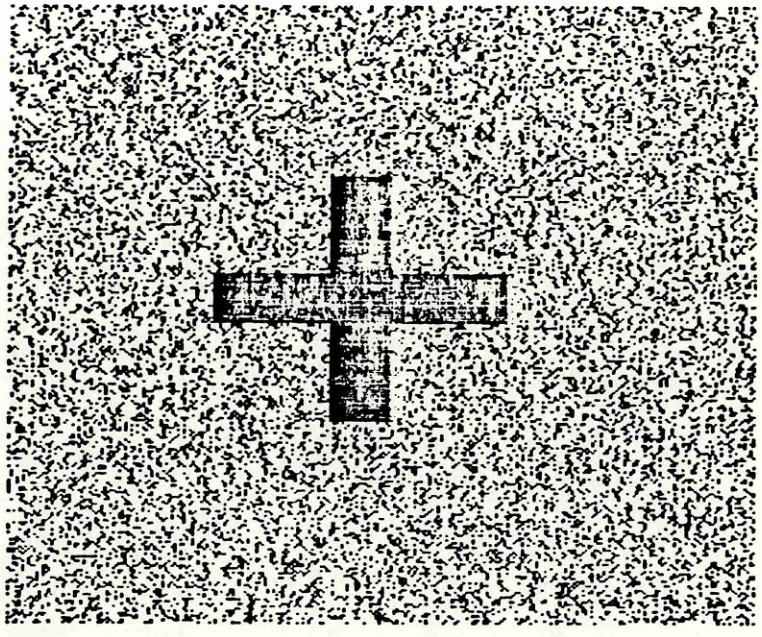


IMAGE 3. Cross with 20,000 random noise pixels

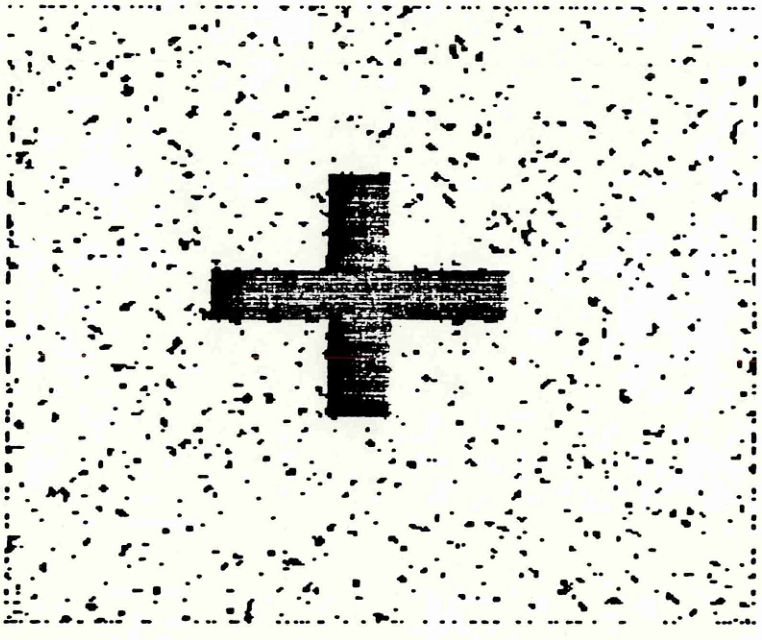


IMAGE 4. Result of IMAGE 3 being processed by a 1 1 1 kernel
1 2 1
1 1 1

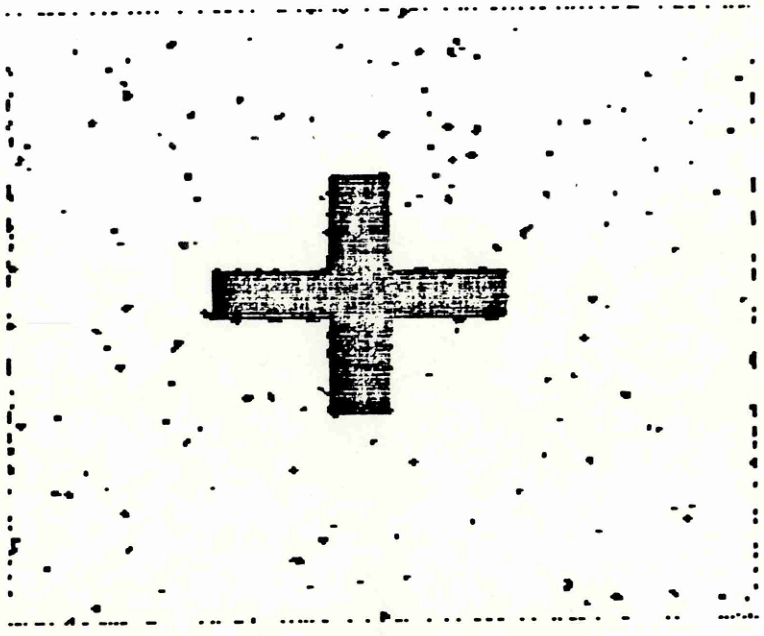


IMAGE 5. Continued processing of IMAGE 3

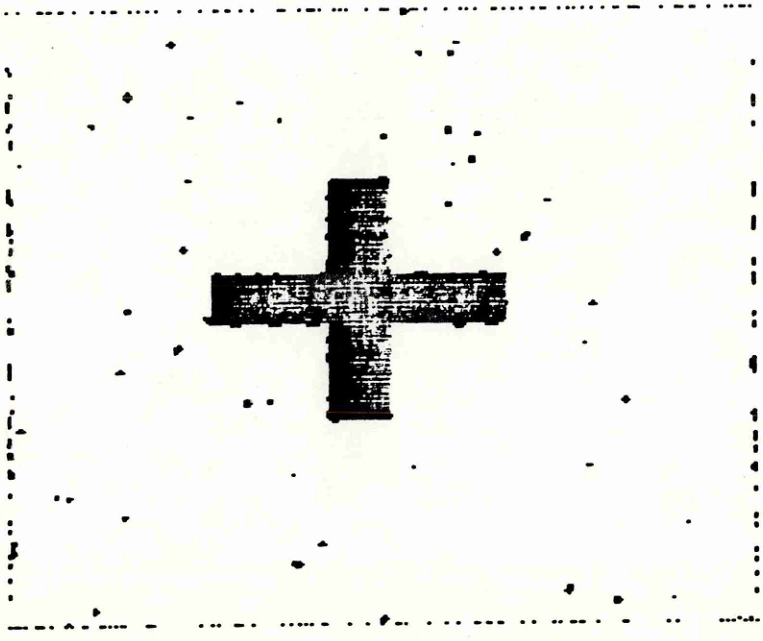


IMAGE 6. Continued processing of IMAGE 3

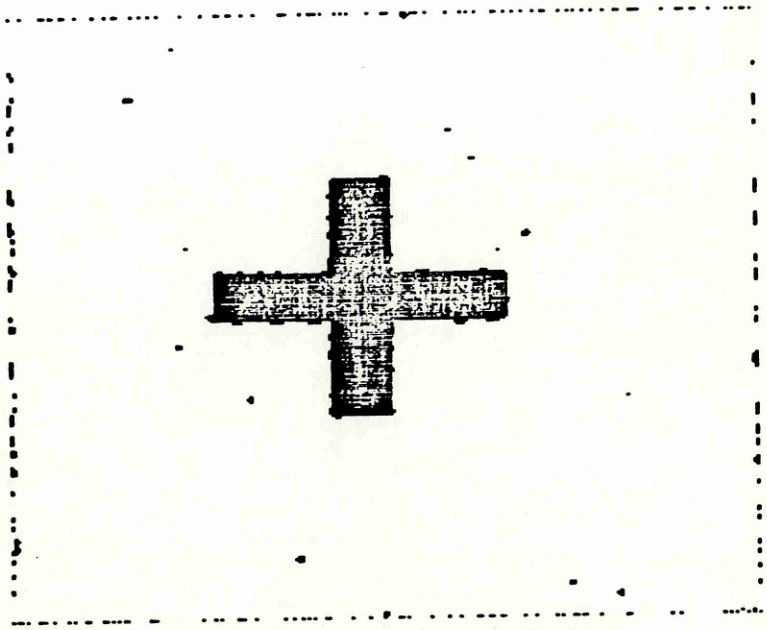


IMAGE 7. Continued processing of IMAGE 3

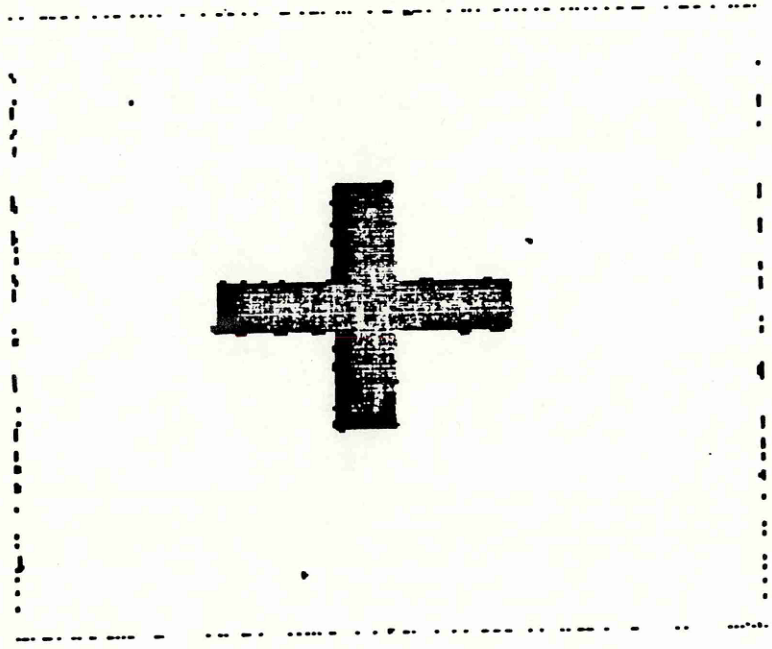


IMAGE 8. Continued processing of IMAGE 3

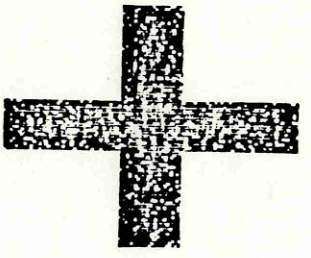


IMAGE 13. Cross with a large number of random noise pixels over the top

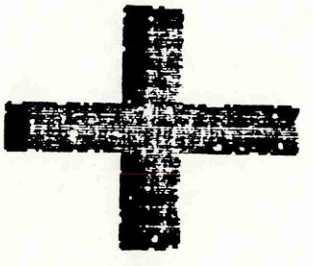


IMAGE 14. Result of IMAGE 13 processed by a
1 1 1
1 2 1
1 1 1
kernel

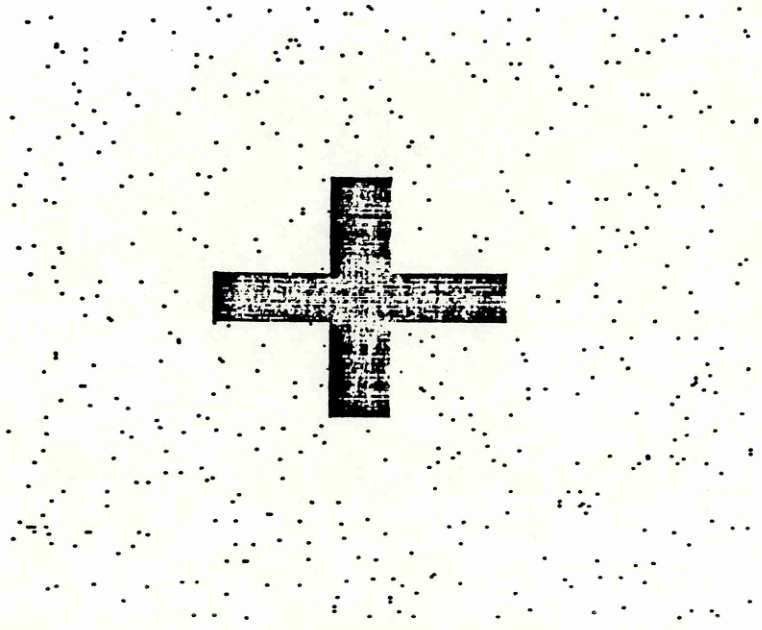


IMAGE 9. Cross with 500 random noise pixels

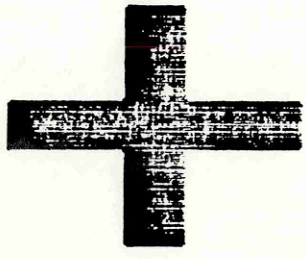


IMAGE 10. Result of IMAGE 9 processed by a 1 1 1 kernel
1 1 1
1 1 1

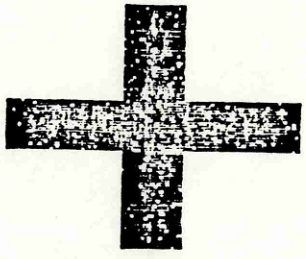


IMAGE 11. Cross with random inverse noise pixels over top

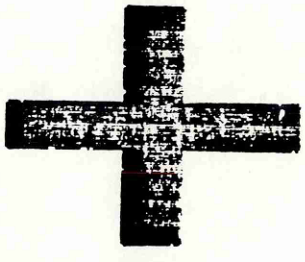


IMAGE 12. Result of IMAGE 11 processed by a $\begin{matrix} 1 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 1 \end{matrix}$ kernel

ACKNOWLEDGEMENTS

My thanks to Professors C.A.R Hoare and D.Hooper for providing data that might otherwise not have been obtainable. Also to A.Finch who constructed the multi-processor array under my supervision as part of his final year degree project.

I must also express my gratitude to those who have acted as supervisors and advisors over the years: Dr P. Thomas at the Open University, Dr M. Taylor at Liverpool University and Dr T. Buckley at Leeds University.